2

CALSPAN-UB RESEARCH CENTER

TECHNICAL REPORT

CALSPAN UB RESEARCH CENTER

CALSPAN-UB RESEARCH CENTER P O BOX 400 BUFFALO, NEW YORK 14225

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY N/A 2b. DECLASSIFICATION / DOWNGRADING SCHEDULE N/A	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S) AFOSR-TR- 89-1752
6a. NAME OF PERFORMING ORGANIZATION Calspan - UB Research Center (CUBRC) 6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION AFOSR/NM
6c. ADDRESS *(City, State, and ZIP Code)* PO Box 400, 4455 Genesee Street Buffalo, NY 14225	7b. ADDRESS *(City, State, and ZIP Code)* Bldg 410 Bolling AFB DC 20332-6448
8a. NAME OF FUNDING / SPONSORING ORGANIZATION Air Force Office of Scientific Research 8b. OFFICE SYMBOL *(If applicable)* NM	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F49620-88-C-0050
8c. ADDRESS *(City, State, and ZIP Code)* Bolling Air Force Base Washington, DC 20332-6448	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. 61102F · PROJECT NO. 2304 · TASK NO A7 · WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
Final Technical Report for the Knowledge-Based Extensible Natural Language Interface Technology Program

12. PERSONAL AUTHOR(S)
Jeannette G. Neal, Ph.D.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Final Technical	FROM 2/88 TO 9/89	30 November 1989	66 Pages

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	Natural Language Processing, Extensibility, Learning
12	05		

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

This research project addressed the problem of developing knowledge-based natural language interface technology that is extensible via natural dialogue between user and computer system. Natural language understanding systems need to be extensible to accomodate changes in the target application system to which they interface as well as to accomodate new users. Typically, however, current systems cannot be extended as part of a normal dialogue session. Instead, extensions must be incorporated and complied into the interface "off line" before the interface is loaded for use. This can be costly in terms of "down-time" and frustrating for the new user.

The solution that was pursued in this project was to develop a natural language interface system, called Lydia, that is extensible via methods that are modeled after human behavior. Specifically, the following methods were implemented in the Lydia system: (a) "learning by being told" including the ability to understand natural language when it is used as its own

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION [illegible]
22a. NAME OF RESPONSIBLE INDIVIDUAL Dr. Abraham Waksman	22b. TELEPHONE *(Include Area Code)* (202) 767-5027 · 22c. OFFICE SYMBOL NM

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

90 01 04 001

Block 19, continued

meta-language to explain new concepts, relations, and rules and (b) being able to infer the category and attributes of new words from their linguistic context when used in a natural language sentence.

Lydia is a prototype proof-of-concept NLP system which consists of the following major components: (1) an extensible knowledge base (KB) designed for the representation of linguistic and application domain knowledge, (2) a Kernel Language and core of predefined linguistic knowledge represented in the KB that provides the system with a knowledge acquisition and bootstrap capability, (3) a parsing procedure that performs syntactic analysis according to the language processing rules represented in the KB, (4) an interpretation procedure that applies the semantic analysis rules represented in the KB, (5) a representation of the dialogue focus space and procedures for its modification and access, and (6) a linguistic inference processor that augments the KB with new linguistic knowledge that is inferred during the parsing/interpretation process.

Keywords: expert system; speech recognition; computer programs; man computer interface; (KR)

2

CALSPAN UB RESEARCH CENTER

Final Report for the Knowledge-Based Extensible Natural Language Interface Technology Program

Jeannette G. Neal

November 29, 1989

Final Report for the Knowledge-Based Extensible Natural Language Interface Technology Program

Jeannette G. Neal

November 29, 1989

1 Introduction

The problem addressed by this project was to develop knowledge-based natural language processing (NLP) technology that is extensible via natural dialogue between user and computer system. Natural language (NL) understanding systems need to be extensible to accommodate changes in the application domain that are reflected in modifications and/or enhancements to application systems and their databases and to accommodate new users who may not easily adapt to the language accepted by the NL interface system. Typically, however, current systems cannot be extended as part of a normal dialogue session. Instead, extensions must be incorporated and compiled into the interface "off line" before the interface is loaded for use. This can be costly in terms of "down-time" for the application system and be frustrating for a user who cannot easily learn the language that the system understands and would like the system to learn some of the language that he wishes to use.

The solution approach used in this project is based upon the view that NLP systems should be extensible via methods that are modeled after human behavior. These methods include: (a) "learning by being told" including the ability to understand a natural language when it is used as its own meta-language to explain new concepts, relations, and rules and (b) being able to infer attributes of new words or phrases from the way they are used by others.

No single learning method will suffice for all situations or conditions. A learning method based on discovery or induction may be appropriate in certain situations, but may be two slow, expensive, and error-prone for other situations in which speed and accuracy of learning is important. Explicit instruction or "learning by being told" may be needed when information must be communicated accurately or when a larger volume of technical information must be learned. Learning methods are frequently used in combination, also. For example, if participant in a two-person dialogue does not know the word "helicopter" and the other person tries to explain the word, the latter person might explain that "a helicopter is a type of aircraft". Now the hearer of this sentence would naturally use both techniques: learning by being told and inference of linguistic attributes of the word "helicopter". The hearer of the explanation not only learns the information explicitly expressed in the sentence (*learning by being told*), but also *infers* certain attributes of the word by filling constraints and/or expectations that are imposed by rules that he uses to parse and understand NL. In this case, these rules include the constituent structure specification rule for a noun phrase and the rule for number agreement between determiner and head noun of a noun phrase. Applying his knowledge that a noun phrase can consist of a determiner followed by a noun, the hearer would infer that the lexical category of "helicopter" is the noun category. Applying his knowledge about number agreement between the determiner and head noun of a noun phrase, he would infer that "helicopter" is singular since it was used with the singular determiner "a". People commonly use combinations of learning methods in this manner. Our approach on this project has been to develop a NL understanding system that is able to use this particular combination of learning methods (i.e., learning by being told as well as learning by inferring the category and attributes of new words from their linguistic context when used in a NL input sentence) to acquire new knowledge about language understanding.

We are particularly interested in a NLP system being able to "learn by being told" about its natural language with the natural language being used as its own meta-language to express/communicate new knowledge about the natural language itself. That is, we are interested in having a NLP system understand natural language when it is used as its own meta-language to express to the system new rules and facts about the natural language that will extend the system's language processing ability. A natural language such as English is commonly used in this manner. Dictionaries use a natural language as its own meta-language to explain the meaning and attributes of the words and idioms of the language. Educational courses in language and linguistics similarly use a natural language as its own meta-language to teach students about the language. And finally, a person engaged in casual conversation may find it necessary to explain the meaning of word, phrase, or grammatical structure that is unknown to the other partner. A simple example was discussed above with the word "helicopter" being the unknown word being explained.

Just as people can understand natural language when it is used as its own meta-language to explain new rules and facts of the language, a NLP system should be able to do likewise. This would enable the user of a computer system to explain what he means to the language processing system when caught in the situation of having used a language structure, word, or phrase that the system is not able to process. The user should be able continue with his task without losing significant time in recovering from an error situation or in taking the system "off line" to revise and recompile its grammar and/or lexicon to adapt it to a new user.

As part of this project, a prototype proof-of-concept NLP system, called Lydia, has been developed. Lydia is an NLP shell consisting of the following major components: (1) an extensible knowledge base (KB) designed for the representation of linguistic and application domain knowledge, (2) a Kernel Language (KL) and core of predefined linguistic knowledge represented in the KB that provides the system with a knowledge acquisition and bootstrap capability, (3) a parsing procedure that performs syntactic analysis according to the language processing knowledge represented in the KB, (4) an interpretation procedure that applies the semantic analysis rules represented in the KB, (5) a representation of the dialogue focus space and procedures for its modification and access, and (6) a Linguistic Inference Processor that augments the KB with new linguistic knowledge that is inferred during the parsing/interpretation process.

Lydia is an NLP shell, analogous to an expert system shell. NLP knowledge is represented in Lydia's knowledge base just as application-domain knowledge is in an expert system. The Kernel Language (KL) is provided as part of Lydia's core for the purpose of knowledge acquisition. This KL provides for the input of rules and facts to define the syntax, features/functions, semantics, and pragmatics of the natural language to be understood and used by Lydia for human-computer dialogue. The KL enables a systems developer or "teacher-user" to build the system to a point where new knowledge about NL processing can be input in the NL itself. Lydia's NLP knowledge and ability can continue to be extended: new NLP knowledge can be input at any time using the KL or the natural language that she is able to understand at the time. Thus, for example, just as a person engaged in conversation with another person can introduce and explain new words to his dialogue partner, Lydia can accept new knowledge about her NL which is also expressed in the NL itself during an application-oriented dialogue.

In combination with this unique ability to understand natural language when used as its own meta-language to explain new concepts, words, and language processing rules, Lydia can infer the category and attributes of new words from their linguistic context when used in NL input sentences. This ability is provided by the Lydia's Linguistic Inference Processor.

The next section of this report discusses related research and background for this project. Section 3 includes the statement of work (SOW) as listed in the Contract as Item 0001AA of PART I, SECTION B. Section 4 discusses the technical accomplishments on the SOW tasks for this project. The subsections of Section 4 are organized so that each subsection addresses a task of the SOW. The particular task addressed by each subsection is stated in italics at the beginning of the subsection. Section 5 discusses publications, Section 6 professional personnel associated with the project, Section 7 interactions, and Section 8 covers inventions and patents. Section 9 provides a summary of this report. The appendices include a description of the KL, in BNF form, along with an example definition, expressed in the KL, of a subset of natural language that was used in a demonstration of Lydia's extensibility and functionality.

2 Background

The research being conducted on this project focuses on (1) "learning by being told" and (2) the inference of linguistic knowledge about new words from the way they are used by others. The following paragraphs briefly discuss these two methods of extending the capabilities of a natural language understanding system, review related work in this area, and indicate how this related work differs from the methodology and system that has been developed on this project.

2.1 Learning by Being Told

Explicit instruction aimed at extending one's command and understanding of a language can be "formal" as in a structured educational language program or it can be "informal" as in the case of a person consulting a dictionary for the meaning, linguistic category, and attributes of a certain word. In either case, this type of learning is usually categorized as "learning by being told".

A significant aspect of this mode of learning a "target language" by instruction is that the target language is frequently used as its own meta-language. Commonly, formal instruction about a language is given in the target language itself (e.g., instruction about English given in English). Similarly, dictionary entries use the target language as its own meta-language to give the meaning and attributes of the words and expressions of the target language.

Explicit instruction is needed when information must be communicated accurately or when ambiguities need to be avoided. It is also necessary when a larger volume of technical information must be taught. Learning by discovery or induction is most appropriate for learning new fundamental concepts or procedures, but for more advanced technical concepts and procedures, this form of learning may be too slow, expensive, and error-prone.

Research on the PARSNIP Natural Language Understanding project [Neal85a, Neal85b, Neal86, Neal87] was concerned with the issue of NL interface extensibility. The focus of the PARSNIP project was on the fundamental requirements and knowledge representations to enable a user of the interface system to bootstrap from a small Core knowledge base using a natural language as its own meta-language. This research focused on determining the knowledge primitives and processing primitives that such a system would initially need to support the bootstrap concept. This current project is an outgrowth of our previous work on the PARSNIP project and builds on some of the results of the PARSNIP project.

As in the case of the current project, the work of Haas & Hendrix [Haas80, Haas83] with the NANOKLAUS system also involves machine learning via dialogue in a subset of NL. In contrast to our approach, however, (1) Haas and Hendrix do not use a methodology that integrates linguistic and domain knowledge in a common representation formalism and/or knowledge base, and (2) Haas and Hendrix do not use a single common natural language for both extending the "application language" and as the application language itself (i.e., they do not integrate the application language and the meta-language for the application language). NANOKLAUS was implemented using LIFER [Hendrix77a, Hendrix77b, Hendrix78]. LIFER consists of two basic parts: (a) a set of interactive language specification functions with which an interface builder defines an application language and (b) a parser which uses the defined application language to parse and interpret inputs expressed in the application language. The first of the two basic parts (the meta-language facility of LIFER) is separate and distinct from the second basic part (the application language). This contrasts with our approach which consists of the system having just one knowledge representation formalism for both linguistic and non-linguistic knowledge, an integrated knowledge base containing linguistic and non-linguistic knowledge, and also having just one language which serves as its own meta-language as well the applications language.

The UNIX Consultant (UC) [Wilensky88] also has a meta-language facility. UC includes a component called the UC Teacher [Wilensky88, Martin87] which accepts input in a subset of English to extend its domain knowledge as well as its language processing ability. The UC Teacher, however, is a separate component of UC with its own knowledge base. This again contrasts with our approach as summarized in the preceding paragraph.

VOX (Vocabulary Extension System) [Meyers85] is a language processing system that is used to process message texts. VOX consists of two subsystems: a text analyzer and an extensibility system. The knowledge representation used is called Conceptual Grammar, encompassing representations of words and phrases, parts of speech, events, and scenarios. The extensibility system lets the user add vocabulary, phrases, events and scenarios to the knowledge base. Our approach differs from the approach taken in the VOX system. We are using a unified system for both text understanding and system extension in contrast to the two separate systems used in VOX. The motivation for the research and the application scenario are at the root of the difference in approach. The objective of the VOX system is to analyze messages while our objective is to provide a natural language interface to a target information system with the interface extensible via natural dialogue, conducted in the system's one natural language.

2.2 Inferring Linguistic Knowledge About New Words

Inference-based lexicon learning techniques are appropriate when other methods of learning the precise meaning of a word are not successful. Thus after the system unsuccessfully tries to look up the word in resources such as its lexicon and a library dictionary (i.e., a dictionary that is not an integral part of the NL processing system) as well as trying to consult a knowledgeable person (perhaps the user), then it could resort to inferring the meaning of a new word or phrase from the context and related words. Jacobs and Zernik [Jacobs88] describe a method for a NL processing system to acquire new words from multiple examples in texts. In this approach, the system uses linguistic and conceptual context to "guess" the general category of a new word, and generate a meaning for the new word via a method of hypothesis formation, refinement, and generalization over multiple examples. Zernik [Zernik89] has developed a method of predicting lexical structures based on a set of lexical categories and combined this with a validation procedure based on scanning text examples. The approach of Jacobs and Zernik focuses on the inference of lexical category and meaning from examples. In contrast, the approach of this research project employs a combination of inferring syntactic category and feature values when possible coupled with the ability to accept explanations in the natural language in which the human-computer dialogue is being conducted.

GENESIS [Mooney85, Mooney87] includes a word learning component which supports acquisition of word meanings and their underlying concepts concurrently. GENESIS is an explanation-based learning system which acquires or learns a plan schema from a single instance by determining why a particular sequence of actions observed in a specific narrative allowed the actors to achieve their goals. Our approach is similar to the extent that our system learns both words and their underlying concepts concurrently. However, as stated in the previous paragraph, our approach is to have the system infer syntactic category and feature values of a new word, but we do not currently attempt to infer meanings. Our focus is on having the system be able to infer attributes and category of new words in a natural manner when the system's natural language is used as its own meta-language to input new linguistic knowledge.

In the preceeding subsections, research into "learning by being told" has been reviewed as well as research into systems/methodologies which enable a system to infer attributes and meanings of new words. This current project addresses the problem of enabling a natural language understanding system to use both inference of linguistic knowledge as well as explicit instruction in a natural combined manner for extension of the system's language understanding ability via human-computer dialogue. This requires that the system be capable of understanding inputs that use the system's natural language as its own meta-language.

3 Statement of Work

The SOW appearing in the Contract as Item 0001AA of PART I, SECTION B, is listed below for convenient reference. This SOW addressed a two-year program, but the Contract funding only provided for one of the two years that were originally proposed.

Task 1. CALSPAN-UB will acquire LOGLISP if available and select between LOGLISP and PROLOG for implementation.

Task 2. A structured inheritance hierarchy will be established to provide an efficient entity-oriented methodology for the representation of linguistic and application domain knowledge in an integrated knowledge base.

Task 3. A new interpretation procedure will be developed that provides the interface system with the ability to use its natural language as its own meta-language.

Task 4. Knowledge representation and natural language processing methodologies will be investigated to enable the interface system to infer information about new unknown words/phrases.

Task 5. The knowledge representation technology, natural language processing methodology, and extensibility of the natural language system will be tested by building up a language definition and knowledge base of application-domain knowledge.

Task 6. A core question answering ability will be developed that can be enhanced via the natural language interface.

Task 7. The natural language processing methodology of the natural language interface system will be extended so that it accepts constrained analogies/comparisons for natural language interface extension.

The technical progress accomplished on each of these tasks is discussed in the next section. Since only one year of the two-year research program was funded, Tasks 6 and 7 have not been addressed.

4 Technical Progress Summary

The approach for this research is based on the view that natural language understanding systems should be extensible via methods that are modeled after human behavior. These methods include:

- "learning by being told" about ones language by having the ability to understand natural language when it is used as its own meta-language tor explain new concepts and rules.
- being able to infer attributes of new words from the way they are used by others.

As part of this project, we have developed a natural language understanding system, called Lydia, which can extend its language ability by using the above specified methods. Figure 1 provides an overview of Lydia's software design. Lydia's major components are:

(a) an extensible knowledge base where both linguistic and application domain knowledge is represented in a declarative and consistent form. A structured concept inheritance hierarchy (Object-Oriented programming in Prolog) serves as the central organizational data structure of the knowledge base.

(b) a core of predefined linguistic knowledge including a Kernel Language that supports *knowledge acquisition* and can be used by the teacher-user to initially instruct Lydia in the processing and understanding of natural language.

(c) a top-down, left-to-right Definite Clause Grammar (DCG) parser that applies the linguistic knowledge represented in the knowledge base.

(d) a bottom-up interpretation routine that produces the representation(s) of the interpretation(s) of the input utterances. The interpretation procedure is dependent on the syntactic, functional, and semantic components of the parsed structure.

(e) a representation of the dialogue focus space and procedures for its modification and access.

(f) a Linguistic Inference Processor that augments the KB with new linguistic knowledge that is inferred during the parsing/interpretation process.

Note that the following notational convention is used in the body of this report: symbols consisting of a character string enclosed in angle brackets (e.g., <PTree> or <word>) represent variables of the descriptive language of this report.

The following subsections address the individual tasks of the SOW.

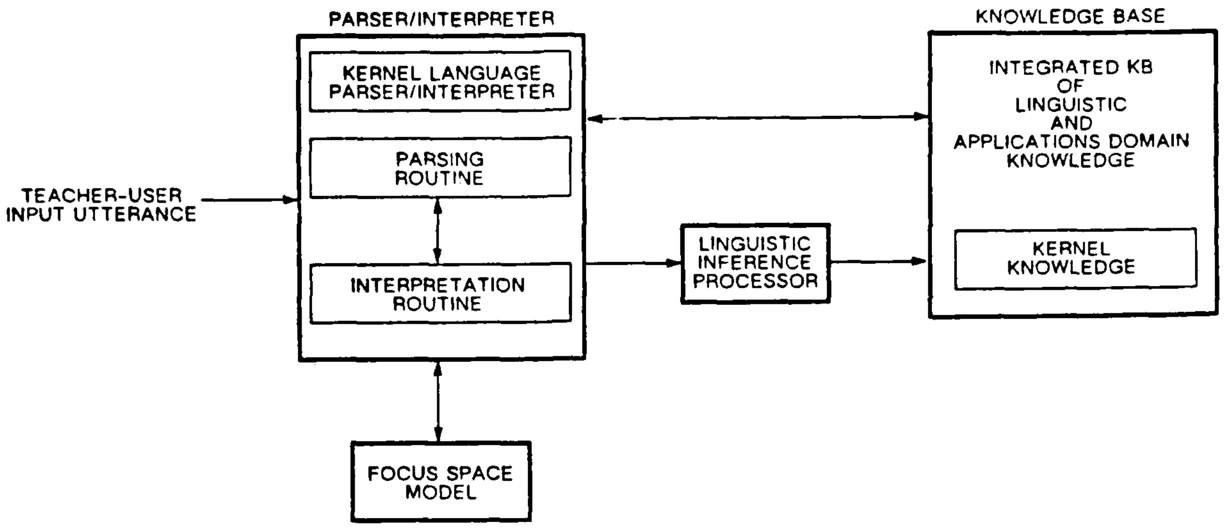
PARSER/INTERPRETER
KERNEL LANGUAGE PARSER/INTERPRETER
PARSING ROUTINE
INTERPRETATION ROUTINE
TEACHER-USER INPUT UTTERANCE
FOCUS SPACE MODEL
LINGUISTIC INFERENCE PROCESSOR
KNOWLEDGE BASE
INTEGRATED KB OF LINGUISTIC AND APPLICATIONS DOMAIN KNOWLEDGE
KERNEL KNOWLEDGE

Figure 1: Overview of Lydia's System Structure

4.1 Implementation Language

Task 1. CALSPAN-UB will acquire LOGLISP if available and select between LOGLISP and PROLOG for implementation.

The AI language LOGLISP [Robinson80, Robinson85] was acquired from RADC and an assessment was made to determine whether LOGLISP or Prolog would be most appropriate for the implementation of the NL processing system designed as part of this project. LOGLISP was originally designed and developed at Syracuse University (partially funded by RADC) and a compiler was developed by Honeywell Inc. under contract with RADC. The decision was made to use Prolog for the prototype NL interface system implementation for the following reasons:

- The typical Prolog language interpreter/compiler requires less main memory and disk storage than does the LOGLISP compiler/interpreter. This is to be expected since LOGLISP provides a logic programming language in addition to Common Lisp.
- There is little to no support for the LOGLISP product at this point. On the other hand, Prolog is available commercially and excellent support is provided by many vendors.
- Prolog interpreter/compilers are available on a wide variety of machines. The current LOGLISP compiler from RADC, however, is designed to run only on VAX computers with the VMS operating system.
- Funds for computer usage in this program were very limited.

The reasons that weighed most heavily in the decision to use Prolog for this program were that: (a) LOGLISP is not a supported product while many commercial software companies provide excellent support for their products and (b) the LOGLISP compiler available from RADC is designed to run only on VAX computers using the VMS operating system. In particular, we selected Arity Prolog from Arity Corporation and SICStus Prolog from the Swedish Institute of Computer Science. Both of these versions of Prolog are high-quality products: in addition to the standard Prolog as defined by Clocksin & Mellish [Clocksin81], they include a large set of additional useful predicates and operators. Arity Prolog is reasonably priced and runs on the family of IBM PC compatibles. The Arity Company provides excellent customer support, including a hot line and an electronic bulletin board. Our version of SICStus Prolog runs on an Encore Multi Max at the State University of New York at Buffalo (SUNYAB) with the UNIX operating system. The combination of Arity Prolog running on IBM PC compatibles and SICStus Prolog on the SUNYAB UNIX machine has provided a functionally rich development environment at low cost and high availability.

4.2 Knowledge Representation

Task 2. A structured inheritance hierarchy will be established to provide an efficient entity-oriented methodology for the representation of linguistic and application domain knowledge in an integrated knowledge base.

4.2.1 Structured Inheritance Hierarchy

Knowledge representation methodology for this project has been developed to meet the goal of the project, namely, extensibility of the natural language used for human-computer communication. In this project, extensibility is to be achieved via natural dialogue between user and computer system. Included among the extensibility methods is the ability of a system to "learn by being told", particularly to understand natural language when it is used as its own meta-language to extend the natural language itself. In order to support this form of extensibility, the language knowledge that the system uses to accept and understand natural language must be part of its domain of discussion and must be represented declaratively in its knowledge base. This language knowledge must be accessible just as any application domain knowledge.

We assume in this project that all forms of language knowledge should be extensible: syntactic, functional, semantic, pragmatic. Therefore, all these types of language knowledge are represented and included in the knowledge base of the system. In particular, since the semantics of language involves the relationship between language and the concepts that the language conveys, it is necessary for the knowledge base to include the representation of both linguistic and non-linguistic knowledge in an integrated manner.

A structured inheritance hierarchy has been developed and implemented as part of this project to provide an efficient object-oriented paradigm for the representation of linguistic and application domain knowledge in an integrated knowledge base. A structured inheritance hierarchy [Touretzky87] based on an object-oriented knowledge representation methodology provides a modular organization of information and eliminates the need to have redundant representation of information that can be inherited via the hierarchy. Our structured hierarchy is implemented in Prolog and is based, in part, on the implementation discussed in [Stabler86] and [Zaniolo86].

The development of the structured hierarchy entailed the design and implementation of the hierarchy data structure with its inheritance and access functions. The hierarchy data structure consists of:

(a) nodes for the representation of concepts,

(b) edges between nodes to depict relations among the concepts represented by the nodes, and

(c) a frame of information associated with each node; the frame consists of slot-value pairs.

The nodes of our hierarchy are classified as *generic* or *individual* [Brachman83]. The generic nodes represent types or classes of entities (e.g., animal, human, noun, sentence) while the individual nodes represent instances of types (or members of classes) in the hierarchy. Generic concepts can be generalizations or specializations of other generic concepts.

The hierarchy is structured via two types of predefined arcs representing two different relations. The first is the *isa* relation which may hold between two entity-types and the *instance_of* relation which associates specific instances of a type and the given entity-type. Instances always appear as leaf nodes of the hierarchy. The *isa* relation is represented by a single line in Figures 2 and 3 while the *instance_of* relation is represented by a double line. The *isa* relation is defined to be a reflexive, transitive relation that provides the hierarchy its inheritance capabilities. The Prolog representation of the *isa* relation between two entity-types is *isa(<entity-type-1>,<entity-type-2>)* and the representation of the *instance_of* relation is *instance_of(<entity-type>,<instance>)*. For example, the Prolog representation of the *isa* relation between nodes n9 and n5 is *isa(n9,n5)*. The Prolog representation of the relation between nodes [verb] and n11 is *instance_of(verb,n11)*.

For this project, since the system must be able to discuss NL using the NL itself, it is important to distinguish between a concept and the word or language which expresses it. Therefore, each node in the hierarchy which represents a concept other than a word itself is represented by a node with a system-generated label consisting of the letter "n" concatenated with a unique integer (e.g. n10). A node in the hierarchy representing any given word is represented by the word itself. In order to allow for multi-word lexemes in our system, we actually use the list containing the (possibly multi-word) lexeme to represent the lexeme itself. In Figure 3, the leaf node labeled with [verb] represents the word "verb" itself, while the node labeled n13 represents the concept of a verb. For convenience, next to each system-generated node name in the figure, we have placed a word which expresses it. The relation

```
                        n1 [entity]
                         /   \
                        /     \
 [abstract entity] n2          n3 [concrete entity]
                  /  \
                 /    \
       [syntax] n5     n4 [feature concept]
             ___/  \___
            /          \
        n9              n10
     [lexeme]        [string]
        /
       /
    n11
 [l-cat]
```

Figure 2: The Kernel Portion of the Structured Hierarchy

between a node such as n13 and the term that expresses it is represented by the *id* (identifier) relation. The first argument of this binary relation is the system-generated node label and the second is the term that expresses it. The actual Prolog representation of this relation is *id(n13, [verb])*.

Associated with each node in our structured concept inheritance hierarchy is a frame of information about the concept represented by the node. Specifically, a frame contains a number of attributes (slots) that are relevant to the concept and that are instantiated (receive values) during the dialogue between Lydia and the teacher-user. In general, the value (filler) of any attribute of the frame of a concept can be a generic concept, an individual concept, or a proposition (e.g., a rule).

In order to support inheritance of slots and slot-value pairs in the hierarchy, we have defined the infix operators \ (backslash) and : (colon) respectively [Zaniolo84, Stabler86]. The Prolog clause:

(a) *<concept>* \ *frame-template(<variable>)* returns the explicit and inherited slots that constitute the frame of the <concept>.

(b) *<concept>:<slot>(<variable>)* returns the value of the explicit or inherited <slot> of the frame-template of the <concept>.

4.2.2 Kernel Knowledge

As stated in the previous section, Lydia is a NLP shell that includes a knowledge base, parser, interpreter, focus space representation module, linguistic inference processor, and a Kernel Language for initially instructing Lydia in the processing of NL. In order for Lydia to become adept in the use of some language by being instructed by a teacher-user in the use of the language, she must start with some language facility. Therefore, we have equipped Lydia with a core knowledge base which we tried to make theory independent and as small as possible. Specifically, the predefined concepts or entity-types include: *entity, abstract entity, concrete entity, feature concept, syntax, lexeme, string,* and *l-cat.* The linguistic entity-types (i.e., *feature concept, syntax, lexeme, string,* and *l-cat*) are used for the representation of the lexicon, grammar, and associated feature and semantic processing knowledge. Specifically, names of lexical classes (e.g., noun, verb) would be classified as instances of the linguistic entity-type *l-cat.* Concepts representing lexical classes would be classified as being sub-types of the linguistic category *lexeme* while concepts representing string classes (e.g., np, vp) would be classified as sub-types of the linguistic category *string.* Non-predefined entity-types are learned by Lydia by instruction or inference during dialogue with a user. Figure 2 illustrates the kernel portion of the structured hierarchy. The edges linking nodes represent the *isa* relation. Figure 3 illustrates a portion of the hierarchy after additional knowledge has been acquired by Lydia via dialogue with a teacher-user. Associated with each predefined

```
                                n1 [entity]
                              /   \
                             /      \
         [abstract entity] n2       n3 [concrete entity]
                          / \         \
                         /    \        \
              [syntax] n5     n4       n17 [aircraft]
                   ---/ \---           \ \-------
                  /          \          \        \
               n9            n10        n18       n19
            [lexeme]       [string]   [bomber]  [tanker]
       ----/ / \ \----        \ \-----
      /      /    \      \      \      \
     n11   n12    n13   n14    n15   n16
 [l-cat] [s-cat] [verb] [noun]  [np]  [vp]
  ||   \\    \\            ||
  ||    \\     \\          ||
  ||      \\     \\        ||
[verb] [noun]  [np]    [aircraft]
```

Figure 3: A Portion of a Structured Hierarchy

concept are certain predefined slots which constitute the core of the frame of the concept (see Section 4.2.1). This means that the existence and semantics of the slots are predefined, but not their values. The predefined slots are:

- *id* is a slot of every entity. Its optional value is the single-word or multi-word lexeme that expresses the concept.
- *mandatory_action* is a slot of every entity. Its optional value is a rule that is applied whenever the concept is involved in a *create_isa_l* or *create_instance_of_l* action (see Section 4.3) being taken as a result of some semantic interpretation.
- *features* is a slot of any syntactic concept. The value of the *features* slot is a list of names of the features of the particular syntactic type. These names have special significance in that they are also the names of additional slots of the particular concept-type. The values of the individual feature slots can be constants or rules that determine the corresponding feature values in the augmented parse tree at runtime.
- *sempred* is a slot of every syntactic concept. The optional value of the *sempred* slot would be a rule for determining the semantics of the corresponding syntactic construct (See Section 4.2.3).
- *semresult* is a feature of every syntactic type. During the parsing process, the semantic interpretation of any given string is stored as the value of the *semresult* feature in the augmented parse tree (see Section 4.2.3).

Other non-predefined slots (and slot-values) are acquired by Lydia via instruction or inference during dialogue with the teacher-user.

4.2.3 Language Processing Knowledge

Language processing knowledge is represented declaratively in Lydia's knowledge base in the form of Prolog rules, organized in a modular manner via the structured hierarchy. The language processing rules are categorized into three types: *syntactic, functional,* and *semantic.* Initially, in order to "educate" Lydia, all three types of rules are input via the predefined Kernel Language. Lydia's parser/interpreter converts the rules into their declarative knowledge base representation. Subsequently, however, after Lydia has learned a sufficient amount about understanding natural language, additional language processing rules and facts can be input in the natural language (e.g., English) itself.

Syntactic rules are used to make lexical entries and define the syntax of input strings. These rules can be input to Lydia via the KL in the form of context-free linguistic rewrite (production) rules as illustrated in Figures 4(a), 5(a), and 6(a). These figures show examples of the three types of syntactic rules. The Prolog KB representation of each of these rewrite rules, as produced by the parser/interpreter, is presented as part (b) of each corresponding figure.

Figure 4(a) shows the rule type for creating new lexical categories. The rule of Figure 4(a) has two results: it adds the term "verb" to the category called "l_cat" and it creates a new lexical category called "verb"; this latter action is a result of the special status of the l_cat category: whenever any term is added to this class, a new category is established with the new term as its name and this new category is linked into the hierarchy as a subtype of the category called "lexeme". Figure 4(b) shows the Prolog KB representation for the interpretation of the rule entered in Figure 4(a).

(a) l_cat − > "verb".

(b) *instance_of(verb,n11).*
isa(n13,n9),
id(n13,[verb]).

where n9, n11, and n13 represent the concepts lexeme, l_cat, & verb respectively.

Figure 4: Example Rule Creating a New Lexical Category

The rule of Figure 5(a) is essentially a lexical entry. That is, the word "is" is entered into the KB as an *instance_of* a verb. This is equivalent to its being entered as a member of the verb category.

(a) verb − > "is".

(b) *instance("is",n13).*

where n13 represents the concept of a verb.

Figure 5: Example Lexical Entry Rule

The rule of Figure 6(a) creates a new production rule defining the syntax of a new phrase type. This rule creates a new string category called "vp" and adds the rule to the knowledge base. This new category is created with an *isa* relation to the concept called "string" (see Figure 3).

It should be noted that syntactic rewrite rules for string non-terminals, such as the rule of Figure 6(a), are translated into definite clause rules [Pereira80] except for the lexeme non-terminals. Specifically, Lydia translates every lexeme non-terminal appearing on the right

```
(a) vp – > verb np.

(b) vp([vp,[V,NP],Ag_ATree]) -- >
        lex_lookup(verb,V), np(NP),
        {build_Ag_ATree(n20,[V,NP],Ag_ATree)]}.

    where n20 represents the concept of the verb phrase
    type defined by vp – > verb np.
```

Figure 6: Example Rule Defining the Syntax of a String Type

side of a rule that defines the syntax of a string type (e.g., the verb non-terminal on the right side of the rule of Figure 6) into a generic *lex_lookup* predicate (see Figure 6(b)) which has three options: succeed, fail, or hypothesize. Lydia uses this representation to infer information about new unknown words (i.e., their lexical category and their features). This differs from typical Prolog-based parsers such as those of [Pereira80] and [McCord85,McCord87] which cannot perform this hypothesis step since their Prolog version of a grammar rule includes a predicate specifying the linguistic category of each word or phrase in sequence. These predicates have only two options: succeed or fail. Furthermore, Lydia always adds the *build_Ag_ATree* predicate to the end of every string non-terminal rewrite rule (see Figure 6(b)). The *build_Ag_ATree* predicate is the function that performs functional and semantic processing of each natural language input phrase as soon as it has been syntactically recognized (see Section 4.3).

Functional rules define the functional (feature) values for any given syntactic component of a syntactic analysis (parse tree) during the parsing process. For any feature of a string type or lexeme, its value can be a constant, a list of feature-value pairs, or a rule (the application of which occurs at runtime and determines the value of the feature). Such a constant, list, or rule is stored as the value of the corresponding slot (named by the particular feature) of the frame associated with the concept of the string type or lexeme in the structured hierarchy. Figure 7 includes both the teacher-user's input in the KL of assigning the constant *present* to the *tense* feature of the word "is", and its Prolog representation. Figure 8 illustrates a KL rule that defines the *tense* feature of a type of verb phrase consisting of a verb followed by a noun phrase. The Prolog representation of the rule is also shown in Figure 8.

Functional rules are applied in a manner that is highly integrated with the syntactic rules. Specifically, each time a syntactic rule (such as the vp rule of Figure 6) is successfully applied in the parsing process, the associated functional rules are applied to augment the parse tree with appropriate feature values.

Semantic rules are rules for determining the semantic interpretation of the different string types and lexemes. These rules are stored in the *sempred* (semantic predicate) slot of the frame for a given string type or individual lexeme. During language processing, the semantic

(a) KL statement declaring that the word "is" has present tense:

```
tense("is", present).
```

(b) The Prolog KB representation of (a):

```
tense(is,n13,n23).
```

where n13, n23 represent the concepts of verb and present, respectively.

Figure 7: Example Feature Definition in the KL and its Interpretation

(a) KL rule defining the tense of the preceding vp:

```
tense(vp,v_out) => if constituent(vp,verb,v_verb)
                           find_f_value(v_verb,tense,v_f)
                      then eq(v_out,v_f).
```

(b) The Prolog KB representation of (a):

```
tense(n20,CATree,T_Value)
    :- ifthen(( constituent(CATree,verb,Verb),
                find_f_value(Verb,tense,F_Value)),
                eq(T_Value,F_Value)).
```

where n20 represents the concept of the verb phrase type defined by: vp − > verb np.

Figure 8: Example KL Rule for Tense and its Prolog KB Representation

interpretation of a given substring of the input string (derived by applying the appropriate semantic rule) is stored as the value of the *semresult* feature in the augmented parse tree (see Section 4.2.4) for the given string. Figure 9 depicts a KL rule that defines the semantics of one sense of the word "is". The rule says that if the subject and the object of the sentence have specificity indefinite then create an *isa* link between the concepts. All variables of the KL are prefixed by "v_". Figure 9 also includes its Prolog representation. The *eq* predicate succeeds if its arguments can be unified. The *constituent* and *find_f_value* predicates are described in Section 4.2.4. This sense of the word "is" would be used in the interpretation of the input sentence 'a fighter is an aircraft' used to extend Lydia's application domain knowledge.

(a) A KL rule defining one sense of the word "is".

```
sempred("is", if find_f_value(s,subject,v_subj)
                 find_f_value(v_subj,specificity,v_f1)
                 beq(v_f1,indefinite)
                 find_f_value(s,object,v_obj)
                 find_f_value(v_obj,specificity,v_f2)
                 beq(v_f2,indefinite)
             then
                 find_f_value(v_subj,semresult,v_f3)
                 find_f_value(v_obj,semresult,v_f4)
                 create_isa_l(v_f3,v_f4,v_r)
                 return(v_r) ).
```

(b) Prolog representation of the KL semantic rule of (a).
n13 represents the concept of verb.

```
sempred(is,n13,rule1).
rule1(CATree,ATree,Value)
   :- ifthen( (find_f_value(ATree,subject,A),
                 find_f_value(A,specificity,B)),
                 beq(B,indefinite)),
        ifthen( (find_f_value(ATree,object,C),
                 find_f_value(C,specificity,D)),
                 beq(D,indefinite)),
        ifthen( (are_instantiated([B,D]),
                 find_f_value(A,semresult,E),
                 find_f_value(C,semresult,F),
                 create_isa_l(E.F,Value) ),
                 true).
```

Figure 10: Example KL Semantic Rule and its Prolog KB Representation

4.3 The Parsing and Interpretation Process

Task 3. A new interpretation procedure will be developed that provides the interface system with the ability to use its natural language as its own meta-language.

The Lydia NLP system that we have developed has the unique ability to understand natural language when used as its own meta-language. The critical characteristics of the Lydia system that are essential to this ability are:

- Linguistic knowledge is part of Lydia's task domain knowledge. It is represented in a manner that is consistent with application domain knowledge and it is represented in the same KB.
- The knowledge representation used for Lydia's KB distinguishes between any given word (or phrase) and the meaning of the word (or phrase). This makes it possible for Lydia to understand the difference between the use and mention [Quine51] of words and phrases.
- Via Lydia's interpretation procedure, the knowledge structures and data structures that Lydia uses for language processing are accessible via natural language input. That is, the procedures that access and modify the structures can be activated as a result of NL interpretation. The primary knowledge structures and data structures are the integrated KB of linguistic and application domain knowledge, the augmented parse tree that is built for any input sentence during the parsing/interpretation process, and the focus space representation.

The combination of the above features gives Lydia the ability to understand natural language input that uses natural language as its own meta-language to define new language processing concepts and procedures. This ability will be demonstrated via an example session with Lydia in Section 4.4.

The Lydia NLP system includes three processes: syntactic analysis, feature/functional analysis, and semantic interpretation. The runtime operation of the three processes is interleaved: feature analysis and semantic interpretation of any parsed input string is performed as soon as possible (i.e., as soon as the required information is available to satisfy the conditions of the feature and semantic analysis rules). It is well recognized that the semantic interpretation of a word/phrase is sometimes necessary to disambiguate syntactic analysis as in the example "John saw the man on the hill with a telescope". Therefore, in agreement with [Lytinen86] and others, we believe that semantic interpretation (as well as feature/functional analysis) of any given phrase should be performed as soon as possible according to the rules of the system. The feature and semantic information is then available to be used in the analysis of the remainder of the input string, if needed.

Our system includes a representation of the attentional discourse focus space [Grosz78, Sidner83, Grosz85, Grosz86] along with predicates for its access and modification. The attentional discourse focus space representation is a key knowledge structure that supports continuity and relevance in dialogue. This focus space representation is critical for the interpretation of anaphoric references (e.g., pronouns) and certain definite references. The focus space representation is discussed in detail in Section 4.3.2.

Our parsing (syntactic analysis) routine is a resolution-based theorem prover as discussed in [Pereira80], [McCord85], and [McCord87]. However, since our design is driven by the need to support extensibility, our parser differs significantly from those of [Pereira80], [McCord85], [McCord87], and others, in the following:

(a) the ability to infer information about new words. By translating every lexeme non-terminal of a string non-terminal rewrite rule into the generic *lex_lookup* predicate (see Section 4.2.3), we provide our parser with the capability to infer the lexical category of new unknown words and their feature values from the way they are used within an input sentence (see Section 4.4).

(b) the ability to handle all possible features for any given syntactic predicate (e.g., noun, np) even though the features are not known in advance. In contrast to other Prolog-based parsers (e.g., [Pereira80], [McCord85], [McCord87]), we do not use a different parameter per feature for any given syntactic predicate. Extensibility precludes the latter since it would require that the system know all the possible features in advance so as to allocate the correct number of parameters when the syntactic rules are initially input to the system. We avoid this restriction by storing in the parse tree the list of feature-value pairs for each syntactically recognized substring of the input string and by adding the *build_Ag_ATree* to the end of every syntactic rule when it is initially input to Lydia (see Section 4.2.3).

During the parsing routine, as soon as each phrase is syntactically recognized, feature analysis is performed on the given phrase. This is accomplished by the *build_Ag_ATree* predicate (see Figure 6(b)) that initiates application of the appropriate functional rules that are stored declaratively in the knowledge base (refer to Figure 8). The result of the feature analysis is the addition of the list of feature-value pairs to the augmented parse tree for the given string.

Similarly, semantic interpretation is performed on each syntactically recognized substring of the input string as soon as feature analysis is performed. This process consists of applying the semantic rules (discussed in Section 4.2.3) whose conditions are satisfied by the results of the syntactic and feature analysis. A representation of the semantic interpretation is stored as the value of the semresult feature in the augmented parse tree for the given string.

Semantic interpretation of a complete input sentence will result in some action being taken by Lydia. For now, the relevant actions for which we have implemented action predicates to carry them out are the following:

(a) adding new concepts, including linguistic concepts, to the structured hierarchy. This action is carried out by the ternary predicate *create_isa_l(<c1>,<c2>,<r>)*. Given the concepts <c1> and <c2>, *create_isa_l* creates an *isa* link or relation between them and returns the relation as the value of the variable <r>.

(b) adding new concept instances to the structured hierarchy. This includes the addition of new *linguistic concepts* to the hierarchy. This action is performed by the predicate *create_instance_of_l(<c1>,<c2>,<r>)*. Given the concepts <c1> and <c2>, *create_instance_of_l* creates an *instance-of* link or relation between them and returns the relation as the value of the variable <r>.

(c) defining new slot values for new lexical entries. This action is performed by the action predicate *create_slot_lex_fv(<word>,<c2>,<c3>,<r>)*. Given a word <word> and the concepts <c2> and <c3>, *create_slot_lex_fv* creates the representation *<feature>(<word>,<c2>,<c3>)*, where <feature> is the feature of the word <word> for which the concept <c3> is a legal value and <c2> represents the lexical category of the word <word>. Furthermore, the representation created by *create_slot_lex_fv* is returned as the value of the variable <r>.

(d) defining new slot values for any concept. This action is performed by the predicate *create_slot_value(<slot>,<c1>,<c2>,<r>)* . Given a slot <slot>, a concept <c1>, and a second concept <c2>, this predicate asserts a clause of the form *<slot>(<c1>, <c2>)* if such a clause does not exist in the KB and returns this new clause as the value of <r>.

(e) creating a new relation. The predicate *create_relation(<relation-name>,<arg-list>,<r>)*. takes a relation name as its first parameter and a list of arguments as its second parameter. It creates the Prolog clause with the relation name as its functor and the argument list as its parameters. The resulting new clause is returned as the value of <r>.

(f) creating new language processing rules. There are four predicates that perform this type of action: *create-rule, create-rule1, create-rule2, create-rule3.* These predicates are described in Appendix A.

Additional new action primitives can easily be developed and added to the system.

4.3.1 The Representation of the Augmented Parse Tree

During the analysis of any given input string, the syntactic, functional, and semantic information that Lydia derives for the input string is represented in the form of an augmented parse tree. Figure 12 illustrates such an augmented parse tree for the sentence '"aircraft" is a singular noun.' Each node of the tree has three components: (1) the syntactic category of the parsed string, (2) the list of the augmented parse trees for the children, and (3) the list of the feature-value pairs for the parsed string. This representation has been designed to support extensibility and the implementation of a wide variety of linguistic processing theories via this NL "bootstrap" system. Specifically, the representation of feature-value pairs in list form in the parse tree for any input string avoids the necessity of predefining the number of features and the specific features for the system. Each different string type can have a different number of features and, in fact, the features of any given type of string can change as a dialogue with Lydia proceeds. For example, Lydia may initially be taught that verb phrases have only tense and she will process sentences accordingly. Subsequently, Lydia may be taught that verb phrases also have linguistic number, voice, etc. As Lydia learns about each of the features of a verb phrase along with rules that define these features for a verb phrase, Lydia will immediately begin to use the new knowledge for all subsequent sentences that are input to her. As stated above, this representation supports a wide variety of linguistic theories. The particular theoretical approach that has been used in testing Lydia is Lexical Functional Grammar (LFG) [Kaplan82].

We have defined the following Kernel Language predicates that serve as the accessing functions of the augmented parse tree data structure:

(a) the ternary predicate *constituent*(<sub-tree>,<syn-cat>,<atree>). Given an augmented parse tree <sub-tree> and a syntactic category <syn-cat>, *constituent* returns the list of feature-value pairs <atree> associated with the syntactic category <syn-cat>. This predicate performs a breadth first search through the augmented parse tree <sub-tree> and returns the first <atree> corresponding to the specified syntactic category.

(b) the ternary predicate *right-constituent*(<sub-tree>,<syn-cat>,<atree>). This predicate is like the predicate *constituent*. The difference is that this predicate returns the *last* <atree> corresponding to the specified syntactic category after a breadth first search through the parse tree <sub-tree>.

(c) the ternary predicate *find_word*(<sub-tree>,<syn-cat>,<word>). This predicate performs a breadth first search through the <sub-tree> and returns the first word in the tree that is of the specified syntactic category. The word is returned as the value of <word>.

(d) the ternary predicate *find_f_value*(<atree>,<feature>, <value>). Given a list of feature-value pairs <atree> and a feature <feature>, *find_f_value* returns the value <value> that the feature <feature> has in the feature-value pairs list <atree>.

(e) the ternary predicate *find_c_value*(<tree>,<syn-cat>,<value>). Given an augmented parse tree <tree> and a syntactic category <syn-cat>, *find_c_value* (read find constituent value) does a depth-first search of the augmented parse tree <tree>, and returns as the value <value> the second component of the first node of the <tree> it finds which has as its first component the syntactic category <syn-cat>.

4.3.2 Attentional Focus Space Representation

Continuity and relevance are key factors in discourse. Without these factors, people find discourse disconcerting and unnatural. The attentional discourse focus space representation [Grosz78, Sidner83, Grosz85, Grosz86] is a key knowledge structure that supports continuity and relevance in dialogue. This type of knowledge structure is typically used by language understanding systems to determine the referent of anaphoric references, including pronouns, and definite references.

The Lydia system includes a data structure which represents the attentional discourse focus space. This data structure, which we call the *focus list*, is an ordered list of discourse objects. The discourse objects are the objects (i.e., objects, concepts, and propositions) under discussion in the dialogue between the user and Lydia. They are the referents or interpretations of the natural language sentences or phrases that are input by the user and processed by Lydia. Each object on the focus list has an associated weight to represent its saliency in the current focus space.

The Lydia system also provides the teacher-user with the ability to specify rules that govern Lydia's addition and removal of discourse objects to and from the focus list.

Each element of the focus list is a quadruplet consisting of:

1. a representation of the discourse object,
2. the gender of the object, if appropriate and known,
3. the linguistic number of the object, if appropriate and known, and
4. a weight representing the object's importance in the focus space.

The representation of the discourse object would be that of a node or logical form expressed in the representational language used for Lydia's knowledge base. The Kernel Language enables the teacher-user to specify the weights that should be assigned to the discourse objects based on the part of speech of the phrase used to reference the object. Thus the teacher-user may specify different weights for objects depending on whether they were used as the object, indirect object, subject, etc. of a sentence. If the teacher-user specified no weights, then Lydia uses a predefined default assignment. When an object is first mentioned, it is assigned a certain focus weight, and then as subsequent sentences are processed, the object fades in importance or weight. Lydia decrements the weight of each object on the focus list by a certain factor (call it the *fade-factor*; its default value is 10) each time a new sentence is parsed and a new set of objects is added to the focus list. Each object is dropped from the focus list after a certain constant number (call it the *in-focus-count*; its default value is 10) of sentences have been processed by Lydia subsequent to the time of the object's being first added to the focus list. Thus each object remains on the focus list during the processing of ten (using the default number) sentences, at which time it is removed and at which time its weight (importance) would have decreased significantly. An object may, of course, be added to the focus list again if it is referenced again. In fact, this frequently occurs. The values of both the *fade-factor* and the *in-focus-count* can be set by the teacher-user via the

KL.

For each input utterance or sentence, Lydia adds discourse objects to the focus list after having completely processed the utterance. This is done so that the part of speech can be used to determine the weight assigned to each discourse object when it is added to the focus list. This is adequate to handle the interpretation of anaphoric references and definite references that refer to objects mentioned in previous sentences, but it is inadequate to handle the interpretation of anaphôric references and definite references that refer to objects mentioned earlier in the current sentence being processed. For example, suppose that the system is processing the following sentence: "The number of a noun phrase is the number of the constituent head noun of the noun phrase." The phrase "the noun phrase" that appears at the end of this sentence refers to the noun phrase mentioned earlier in the same sentence. This phrase would be mis-interpreted if the system were relying on the focus list, as described above, for resolution of such definite references. The focus list is the right type of mechanism for handling the interpretation of such phrase types, but the interpretation of the first mention of "noun phrase" is not yet on the focus list when the definite reference to the noun phrase is being processed. This problem is solved in the Lydia system by using a "temporary" or "local" focus list. This temporary focus list holds the discourse objects that have been referenced in the sentence being currently processed by Lydia. Entries are made to this temporary focus list at the completion of each phrase, instead of at the completion of each sentence, as in the case of the main focus list. When entries are made to the temporary focus list, the discourse object is added with its gender and number, if available, but no weight is used. The use of a temporary focus list also facilitates adding discourse objects to the main focus list. When Lydia has completed the processing of the sentence and performs the addition of the discourse objects to the main focus list, Lydia uses the temporary focus list as the source of discourse objects to add to the main focus list. For each object on the temporary focus list, a weight is assigned and the object is added to the main focus list. The temporary focus list is reset to an empty status after the transfer of discourse objects is made from the temporary to the main focus list.

The KL provides the teacher-user with the ability to specify rules to govern Lydia's behavior with regard to adding and removing discourse objects to and from the focus list. The following predicates of the KL serve as the accessing functions for the focus list data structure.

(a) the ternary predicate *add-to-focus-list*(<object>,<number>,<gender>). Given a discourse object, the linguistic number of the lexeme used to reference the object, and the gender of the lexeme, *add-to-focus-list* adds the object to the focus list with designated number and gender.

(b) the predicate *on-focus-list*(<object>,<number>,<gender>,<value>). Given a discourse object, the linguistic number of the lexeme used to reference the object, and the gender of the lexeme as parameters, *on-focus-list* returns, as <value>, the matching concepts that are on either the temporary or main focus lists. The temporary focus list is searched before the main focus list.

(c) the predicate *first-on-focus-list*(<object>,<number>,<gender>,<value>). *first-on-focus-list* performs the same procedure as the predicate *on-focus-list* except that *first-on-focus-list* returns only the first discourse object from the focus list (temporary or main) that matches the input parameters.

Using the above listed predicates, the teacher-user can input rules, expressed in the KL, to specify Lydia's behavior with regard to adding and retrieving discourse objects to and from the focus list. The first KL rule of Figure 11 specifies that if the determiner of a noun phrase has indefinite specificity, then add the discourse object to the focus list with its linguistic number. The second KL rule of Figure 11 specifies that if a noun phrase is a definite noun phrase and the discourse object is on the focus list, then use that object as the interpretation of the noun phrase and re-add it to the focus list (it has been referenced again, so keep its saliency high in the focus space).

```
sempred(np,g,if constituent(np,determiner,v_det)
                find_f_value(v_det,specificity,v_sp)
                constituent(np,noun,v_noun)
                find_f_value(v_noun,semresult,v_out)
                beq(v_sp,indefinite)
            then add_to_focus_list(v_out,v_n,v_g)
                return([[],[v_out]]) ).

sempred(np,g,if constituent(np,determiner,v_det)
                find_f_value(v_det,specificity,v_sp)
                beq(v_sp,definite)
                constituent(np,noun,v_noun)
                find_f_value(v_noun,number,v_n)
                find_f_value(v_noun,semresult,v_out)
                first_on_focus_list(v_out,v_w,v_g,v_r)
            then add_to_focus_list([[v_r,v_out]],v_n,v_g)
                return([[],[v_r]]) ).
```

Figure 11: Example KL Rules That Specify Focus List Access

4.4 Inferring Information About New Words/Phrases

Task 4. Knowledge representation and natural language processing methodologies will be investigated to enable the interface system to infer information about new unknown words/phrases.

The knowledge representation and natural language processing methodology discussed in Sections 4.2 and 4.3 has been designed to enable Lydia to infer information about new unknown words/phrases. The critical principle that we have designed into Lydia is that a grammar should be capable of being used to infer information about new words/phrases as well as being used for parsing NL input. Our methodology enables Lydia to infer the lexical category of new words and their feature values from the way they are used within an input sentence (assuming the user is a "teacher-user" and inputs correct language).

Lydia's ability to infer the lexical category of new unknown words as well as the features of the new unknown words is a result of the fact that each syntactic rule is applied in a top-down goal-directed manner by the Prolog inference engine. Thus, for any syntactic rule being applied, Lydia has an expectation as to what linguistic category the next word should belong to as well as what features should be associated with the word (the latter assuming the teacher-user has previously input to Lydia some features of the linguistic category). For example, suppose Lydia knows that the linguistic features of the linguistic category *noun* are *root* and *number*. Now, suppose that Lydia is applying the syntactic rule *np* $->$ *det adj noun* to the phrase "the enemy helicopter" and has successfully parsed "the" as a *det*, and "enemy" as an *adj*. Then, Lydia expects the next word to be a *noun*. If "helicopter" is an unknown word, then Lydia, via the *lex_lookup* predicate (see Section 4.2.3), hypothesizes that "helicopter" is a *noun* with the linguistic features appropriate to a noun according to Lydia's knowledge base. (Note that the linguistic features that are associated with a given syntactic category will change as Lydia's language knowledge grows via dialogue with the teacher-user.) If, according to Lydia's knowledge base, the features of a noun are *root* and *number*, then this hypothesis is represented as the Prolog list: [*noun*,[helicopter],[[*root*,X],[*number*,Y]] which is returned by the *lex_lookup* predicate and is stored in the augmented parse tree that is assembled during the parsing process. It should be noted, that the fact that the values of *root* and *number* are unknown at this point is represented in the augmented parse tree by having uninstantiated Prolog variables (i.e., X and Y) serve as their unknown values. The latter supports the process of inferring the values of the features of new unknown words. For example, if Lydia has a functional rule that specifies that the linguistic *number* of the *det* and *noun* of a *np* should be the same (both singular or both plural), then she can infer the linguistic number of the *noun* from that of the *det* or vice versa via the unification of variables by the Prolog subsystem.

After Lydia hypothesizes the lexical category, the linguistic features, and the values of the linguistic features of a new unknown word (i.e., "helicopter"), she continues parsing the rest of the input string. If the parse of the complete input string is successful with the particular hypothesis, then Lydia assumes it to be correct. Next, Lydia's Linguistic Inference Processor traverses the parse tree for all such successful hypotheses and enters them into the knowledge base. On the other hand, if the parse of the input string blocks and the inference engine backtracks to a point at which the hypothesis was made, then the hypothesis is automatically discarded by the unbinding of variables.

4.5 Testing the System

Task 5. The knowledge representation technology, natural language processing methodology, and extensibility of the natural language system will be tested by building up a language definition and knowledge base of application-domain knowledge.

In order to test the system, we have built up a fairly sophisticated grammar of English using the Lydia system, which is included as Appendix D. This grammar includes conjunctions, prepositional phrases, recursive rules, and embedded clauses, for example. A simpler version of the grammar is included as Appendix C and is used for illustrative purposes in the Session with Lydia that is discussed in this section.

The example session with Lydia that we discuss in this section demonstrates that Lydia fulfills the objectives that were initially established for this project. Namely, Lydia demonstrates the ability to learn or acquire new language processing knowledge via the two methods stated in the Introduction Section:

(a) "learning by being told" including the ability to understand a natural language when it is used as its own meta-language to explain new concepts, relations, and rules and

(b) being able to infer attributes of new words or phrases from the way they are used by others.

In this example session, we demonstrate that not only does Lydia acquire new language processing knowledge via these two methods, but that she is able to immediately use the new knowledge in processing subsequent inputs.

We now present a portion of a session we had with Lydia. Initially, Lydia has very primitive knowledge about natural language understanding and the only means of instructing her is with rules written in the Kernel Language. As mentioned in Section 4.2, there are three types of language processing rules, namely, syntactic, functional, and semantic. We use rules of all three types to educate Lydia in understanding natural language to a point such that she can understand inputs about natural language expressed in the natural language itself. That is, Lydia is then able to understand natural language (i.e., English) when it is used as a meta-language to express new knowledge about understanding natural language itself.

More specifically, we input the KL statements that are included as Appendix C so as to build up Lydia's language processing knowledge to a point where we can input new knowledge about language processing in natural language itself, namely English. The KL statements of Appendix C include rules of all three types: syntactic, functional, and semantic. After inputting the KL statements of Appendix C, we enter the following inputs that are numbered and printed in bold print. The result of each of the following inputs is explained in the accompanying paragraph(s). These inputs are entered in the order shown below.

Sentence 1: '"aircraft" is a singular noun.'

Up to this point, the words "aircraft" and "a" were totally unknown to Lydia (i.e., both the concept represented by each word as well as linguistic knowledge pertaining to each word is not in Lydia's knowledge base). Furthermore, the linguistic category and attributes of the word "singular" is unknown to Lydia, but the concept it represents is known to Lydia since it was assigned by the teacher-user during the bootstrapping process to be the value of the *number* feature of the word "noun" (see Appendix C). Analyzing the sentence with her integrated parsing/interpretation routine, Lydia derives the augmented parse tree of Figure 12. From this we observe that Lydia has produced the following Prolog representation of the meaning of the sentence:

> *instance_of(aircraft,n19),*
> *number(aircraft,n19,n21)*

where n19, n21 represent the concepts of noun and singular respectively. The interpretation of the sentence is represented as the value of the *semresult* feature of the non-terminal s in the parse tree of Figure 12. This means that Lydia now knows that the word "aircraft" is a noun, and that its number is singular. This is an example of Lydia's ability to "learn by being told" about her language, by understanding the use of natural language as its own meta-language by which new concepts are explained.

Also, Lydia has inferred that the words "a" and "singular" are a determiner and an adjective respectively (i.e., *instance_of(a,n16), instance_of(singular, n18)*, where n16 and n18 represent the concepts of determiner and adjective respectively). The above Prolog representations of the lexical categories of "a" and "singular" are produced by Lydia's Linguistic Inference Processor (see Section 4.4). The latter is an example of Lydia's ability to infer the linguistic category of new unknown words from the way they are used by others.

KL Input 1: 'features(determiner,[specificity,number]).'

Via this input in the KL we inform Lydia that the concept represented by the word "determiner" has two features, namely, *specificity* and *number*. The parser/interpreter processes the KL input and produces the following Prolog representation of its meaning:

> *features(n16,[n51,n20]),*
> *isa(n51,n3),*
> *isa(n20,n3),*

where n3, n16, n20, and n21 represent the concepts of feature_concept, determiner, number, and specificity, respectively. It should be remembered that Lydia always creates an *isa* relation between every feature (e.g., *specificity* and *number*) that the teacher-user inputs to her for the first time and the predefined entity-type *feature concept* (see Section 4.2).

```
s <-ATREE-> [
:             [semresult,[ instance_of(aircraft,n19),
:                          number(aircraft,n19,n21) ] ],
:             [sempred,rule3],
:             [subject,[
:                       [semresult,aircraft],
:                       [sempred,rule5],
:                       [modifier,_14D4] ]
:             [object,[
:                      [semresult,n19],
:                      [sempred,rule6],
:                      [modifier,[
:                                 [semresult,n21],
:                                 [sempred,default] ] ] ]
: np <-ATREE-> [
: :              [semresult,aircraft],
: :              [sempred,rule5],
: :              [modifier,_14D4] ]
: : quoted_word_1 <-ATREE-> [
: : :                         [semresult,aircraft],
: : :                         [sempred,rule4] ]
: : : quote ==> " <-ATREE-> [  ]
: : : any_word ==> aircraft <-ATREE-> [
: : : :                                 [semresult,aircraft],
: : : :                                 [sempred,default] ]
: : : quote ==> " <-ATREE-> [  ]
: vp <-ATREE-> [
: :              [semresult,_44AC],
: :              [sempred,rule3],
: :              [object,[
: :                       [semresult,n19],
: :                       [sempred,rule6],
: :                       [modifier,[
: :                                  [semresult,n21],
: :                                  [sempred,default] ] ] ]
: : verb ==> is <-ATREE-> [
: : :                       [semresult,_BA50],
: : :                       [sempred,rule3] ]
: : np <-ATREE-> [
: : :              [semresult,n19],
: : :              [sempred,rule6],
: : :              [modifier,[
: : :                         [semresult,n21],
: : :                         [sempred,default] ] ]
: : : determiner ==> a <-ATREE-> [
: : : :                            [semresult,n50],
: : : :                            [sempred,default] ]
: : : adjective ==> singular <-ATREE-> [
: : : :                                  [semresult,n21],
: : : :                                  [sempred,default] ]
: : : noun ==> noun <-ATREE-> [
: : : :                         [semresult,n19],
: : : :                         [sempred,default],
: : : :                         [number,singular] ]
```

Figure 12: The Augmented Parse Tree of the Sentence: ' "aircraft" is a singular noun'.

KL Input 2: 'features(np,[specificity,number,modifier]).'

The KL language definition of Appendix C that we initially input to Lydia included the statement 'features(np,[modifier])'. This told Lydia that a np has only one feature, namely *modifier*. Now however, we want to add to the features that Lydia has associated with a np by informing Lydia that the features of a np should not only consist of the *modifier* feature but the *specificity* and *number* features as well (since we will next input Sentences 2 & 3 that express rules pertaining to the *specificity* and *number* features of a np). Specifically, after the parser/interpreter analyzes KL Input 2, it deletes from Lydia's knowledge base the Prolog representation of the meaning of the KL input 'features(np,[modifier])' and produces the following representation of the meaning of KL Input 2:

> *features(n26,[n51,n20,n27]),*
> *isa(n51,n3),*
> *isa(n20,n3),*
> *isa(n27,n3),*

where n3, n51, n20, and n27 represent the concepts of *feature_concept, specificity, number,* and *modifier* respectively. Now, Lydia knows that the features of a np are *specificity, number,* and *modifier* rather than just *modifier.*

Sentence 2: 'the specificity of a np is the specificity of the constituent determiner of the np.'

With this natural language input, we demonstrate that Lydia can understand NL input which expresses a new rule about NLP in addition to her ability to understand NL which expresses simpler assertions as demonstrated with Sentence 1. This input also demonstrates that Lydia can infer information about new words from the linguistic context. In this sentence, the word "the" is unknown to Lydia. Also, linguistically the word "specificity" is unknown to Lydia even though the concept it represents is known from the previous two KL inputs (i.e., KL inputs 1 & 2). By applying her syntactic, functional, and semantic rules in an integrated manner, Lydia produces the parse tree of Figure 13. Examining the parse tree, we see that Lydia has inferred that the word "the" is a determiner and the word "specificity" is a noun. The Linguistic Inference Processor produces the Prolog representations of the latter which are:

> *instance_of(the, n16),*
> *instance_of(specificity,n19)*

where n16 and n19 represent the concepts of determiner and noun respectively. Also, by inspecting the value of the *semresult* feature of the non-terminal s, we see that Lydia's understanding of the sentence is the Prolog rule:

```
specificity(NP,CATree,Value)
:-
NP : id(np),
ifthen( ( constituent(CATree,determiner,DET),
          find_f_value(DET,specificity,SPEC)
          ),
          eqq(Value,SPEC) ).
```

Specifically, the rule states that for any np, if DET is the constituent determiner of the np and SPEC is the specificity of DET, then SPEC is the specificity of the np. This is an example of Lydia's ability to "learn by being told" about her language, by understanding the use of natural language as its own meta-language by which new rules are explained.

Sentence 3: 'the number of a np is the number of the constituent determiner of the np and the number of the constituent noun of the np.'

This natural language input shows Lydia's understanding of another sophisticated rule of NLP expressed in natural language. This again demonstrates Lydia's ability to understand natural language when it is used as its own meta-language to explain a new rule of NLP. It also demonstrates Lydia's ability to infer information about new words from the linguistic context. In this sentence, a new unknown word is presented to Lydia, namely, the word "number". Again, as was the case with the word "specificity" of Sentence 2, the concept represented by the word "number" is known by Lydia (from KL inputs 1 & 2) but the *linguistic nature* of "number" is unknown to Lydia. By analyzing the sentence with her integrated parser/interpreter, Lydia infers that the word "number" is a noun, that is, the Linguistic Inference Processor produces the Prolog representation

instance_of(number,n19),

where n19 represents the concept of noun and creates the following Prolog representation of the meaning of the sentence:

```
number(NP,CATree,Value)
:-
NP : id(np),
ifthen( ( constituent(CATree,determiner,DET),
           find_f_value(DET,number,NUM_DET),
           constituent(CATree,noun,NOUN),
           find_f_value(NOUN,number,NUM_NOUN) ),
         ( eqq(Value,NUM_DET),
           eqq(Value,NUM_NOUN) ) ).
```

Specifically, the above Prolog rule states that for any np, if DET is the constituent determiner of the np and NUM_DET is the number of DET, and NOUN is the constituent noun of the

```
s <-ATREE-> [
:             [semresult,[ specificity(NP,CATree,Value)
:                          :-
:                          NP : id(np),
:                          ifthen( ( constituent(CATree,determiner,DET),
:                                    find_f_value(DET,specificity,SPEC)
:                                  ),
:                                  eqq(Value,SPEC) ) ] ],
:             [sempred,rule1],
:             [subject,[
:                       [semresult,[ [specificity(np,v_1)],[v_1]] ],
:                       [sempred,rule10] ]
:             [object,[
:                      [semresult,[
:                                  [constituent(np,determiner,v_2),
:                                   find_f_value(v_2,specificity,v_3)],
:                                   [v_3] ] ],
:                      [sempred,rule10] ] ]
: cnp <-ATREE-> [
: :               [semresult,[ [specificity(np,v_1)],[v_1] ] ],
: :               [sempred,rule10] ]
: : pnp <-ATREE-> [
: : :               [semresult,[ [specificity(np,v_1)],[v_1]] ],
: : :               [sempred,rule7],
: : :               [modifier,_0474],
: : :               [pmod,[
: : :                      [semresult,_15B4],
: : :                      [sempred,_15A4],
: : :                      [pcase,[
: : :                              [semresult,n53],
: : :                              [sempred,default] ]
: : :                      [object,[
: : :                               [semresult,n26],
: : :                               [sempred,rule6],
: : :                               [specificity,_0BE4],
: : :                               [number,_0BFC],
: : :                               [modifier,_0C20] ] ] ]
: : : np <-ATREE-> [
: : : :              [semresult,n51],
: : : :              [sempred,rule6],
: : : :              [specificity,_0438],
: : : :              [number,_0450],
: : : :              [modifier,_0474] ]
: : : : determiner ==> the <-ATREE-> [
: : : : :                               [semresult,n52],
: : : : :                               [sempred,default],
: : : : :                               [specificity,_02E4],
: : : : :                               [number,_02FC] ]
: : : : noun ==> specificity <-ATREE-> [
: : : : :                                [semresult,n51],
: : : : :                                [sempred,default],
: : : : :                                [number,_0360] ]
: : : pp <-ATREE-> [
: : : :              [semresult,_15B4],
: : : :              [sempred,_15A4],
: : : :              [pcase,[
: : : :                      [semresult,n53],
: : : :                      [sempred,default] ]
: : : :              [object,[
: : : :                       [semresult,n26],
¦ ¦ ¦ ¦                      [sempred,rule6],
¦ ¦ ¦ ¦                      [specificity,_OBE4],
¦ ¦ ¦ ¦                      [number,_OBFC],
¦ ¦ ¦ ¦                      [modifier,_OC20] ] ]
¦ ¦ ¦ ¦ preposition ==> of <-ATREE-> [
¦ ¦ ¦ ¦ ¦                             [semresult,n53],
¦ ¦ ¦ ¦ ¦                             [sempred,default] ]
¦ ¦ ¦ ¦ np <-ATREE-> [
¦ ¦ ¦ ¦ ¦             [semresult,n26],
¦ ¦ ¦ ¦ ¦             [sempred,rule6],
¦ ¦ ¦ ¦ ¦             [specificity,_OBE4],
¦ ¦ ¦ ¦ ¦             [number,_OBFC],
¦ ¦ ¦ ¦ ¦             [modifier,_OC20] ]
¦ ¦ ¦ ¦ ¦ determiner ==> a <-ATREE-> [
¦ ¦ ¦ ¦ ¦ ¦                           [semresult,n50],
¦ ¦ ¦ ¦ ¦ ¦                           [sempred,default],
¦ ¦ ¦ ¦ ¦ ¦                           [specificity,_OA90],
¦ ¦ ¦ ¦ ¦ ¦                           [number,_OAA8] ]
¦ ¦ ¦ ¦ ¦ noun ==> np <-ATREE-> [
¦ ¦ ¦ ¦ ¦ ¦                       [semresult,n26],
¦ ¦ ¦ ¦ ¦ ¦                       [sempred,default],
¦ ¦ ¦ ¦ ¦ ¦                       [number,_OBOC] ]
¦ vp <-ATREE-> [
¦ ¦               [semresult,_4458],
¦ ¦               [sempred,rule1],
¦ ¦               [object,[
¦ ¦                        [semresult,[
¦ ¦                                    [constituent(np,determiner,v_2),
¦ ¦                                     find_f_value(v_2,specificity,v_3)],
¦ ¦                                     [v_3] ] ],
¦ ¦                        [sempred,rule10], ] ]
¦ ¦ verb ==> is <-ATREE-> [
¦ ¦ ¦                       [semresult,_1A1C],
¦ ¦ ¦                       [sempred,rule1] ]
¦ ¦ cnp <-ATREE-> [
¦ ¦ ¦               [semresult,[
¦ ¦ ¦                           [constituent(np,determiner,v_2),
¦ ¦ ¦                            find_f_value(v_2,specificity,v_3)],
¦ ¦ ¦                            [v_3] ] ],
¦ ¦ ¦               [sempred,rule10] ]
¦ ¦ ¦ pnp <-ATREE-> [
¦ ¦ ¦ ¦               [semresult,[
¦ ¦ ¦ ¦                           [constituent(np,determiner,v_2),
¦ ¦ ¦ ¦                            find_f_value(v_2,specificity,v_3)],
¦ ¦ ¦ ¦                            [v_3] ] ],
¦ ¦ ¦ ¦               [sempred,rule8],
¦ ¦ ¦ ¦               [modifier,_1CD4],
¦ ¦ ¦ ¦               [pmod,[
¦ ¦ ¦ ¦                      [semresult,_3D2C],
¦ ¦ ¦ ¦                      [sempred,_3D1C],
¦ ¦ ¦ ¦                      [pcase,[
¦ ¦ ¦ ¦                              [semresult,n53],
¦ ¦ ¦ ¦                              [sempred,default], ]
¦ ¦ ¦ ¦                      [object,[
¦ ¦ ¦ ¦                               [semresult,
¦ ¦ ¦ ¦                                [ [constituent(np,determiner,v_2)],
¦ ¦ ¦ ¦                                  [v_2] ] ],
¦ ¦ ¦ ¦                               [sempred,rule9],
¦ ¦ ¦ ¦                               [modifier,[
¦ ¦ ¦ ¦                                          [semresult,n13],
: : : :                                         [sempred,default] ]
: : : :                              [pmod,[
: : : :                                     [semresult,_3684],
: : : :                                     [sempred,_3674],
: : : :                                     [pcase,[
: : : :                                              [semresult,n53],
: : : :                                              [sempred,default] ]
: : : :                                     [object,[
: : : :                                               [semresult,n26],
: : : :                                               [sempred,rule6],
: : : :                                               [specificity,_2CB4],
: : : :                                               [number,_2CCC],
: : : :                                               [modifier,_2CF0] ] ] ] ]
]
: : : : np <-ATREE-> [
: : : : :              [semresult,n51],
: : : : :              [sempred,rule6],
: : : : :              [specificity,_1C98],
: : : : :              [number,_1CB0],
: : : : :              [modifier,_1CD4] ]
: : : : : determiner ==> the <-ATREE-> [
: : : : : :                              [semresult,n52],
: : : : : :                              [sempred,default],
: : : : : :                              [specificity,_1B44],
: : : : : :                              [number,_1B5C] ]
: : : : : noun ==> specificity <-ATREE-> [
: : : : : :                                [semresult,n51],
: : : : : :                                [sempred,default],
: : : : : :                                [number,_1BC0] ]
: : : : pp <-ATREE-> [
: : : : :              [semresult,_3D2C],
: : : : :              [sempred,_3D1C],
: : : : :              [pcase,[
: : : : :                      [semresult,n53],
: : : : :                      [sempred,default] ]
: : : : :              [object,[
: : : : :                       [semresult,[ [constituent(np,determiner,v_2)],
: : : : :                                    [v_2] ] ],
: : : : :                       [sempred,rule9],
: : : : :                       [modifier,[
: : : : :                                  [semresult,n13],
: : : : :                                  [sempred,default] ]
: : : : :                       [pmod,[
: : : : :                              [semresult,_3684],
: : : : :                              [sempred,_3674],
: : : : :                              [pcase,[
: : : : :                                      [semresult,n53],
: : : : :                                      [sempred,default] ]
: : : : :                              [object,[
: : : : :                                       [semresult,n26],
: : : : :                                       [sempred,rule6],
: : : : :                                       [specificity,_2CB4],
: : : : :                                       [number,_2CCC],
: : : : :                                       [modifier,_2CF0] ] ] ] ]
: : : : : preposition ==> of <-ATREE-> [
: : : : : :                              [semresult,n53],
: : : : : :                              [sempred,default], ]
: : : : : pnp <-ATREE-> [
: : : : : :               [semresult,[ [constituent(np,determiner,v_2)],
: : : : : :                            [v_2] ] ],
¦ ¦ ¦ ¦ ¦ ¦               [sempred,rule9],
¦ ¦ ¦ ¦ ¦ ¦               [modifier,[
¦ ¦ ¦ ¦ ¦ ¦                          [semresult,n13],
¦ ¦ ¦ ¦ ¦ ¦                          [sempred,default] ]
¦ ¦ ¦ ¦ ¦ ¦               [pmod,[
¦ ¦ ¦ ¦ ¦ ¦                      [semresult,_3684],
¦ ¦ ¦ ¦ ¦ ¦                      [sempred,_3674],
¦ ¦ ¦ ¦ ¦ ¦                      [pcase,[
¦ ¦ ¦ ¦ ¦ ¦                              [semresult,n53],
¦ ¦ ¦ ¦ ¦ ¦                              [sempred,default] ]
¦ ¦ ¦ ¦ ¦ ¦                      [object,[
¦ ¦ ¦ ¦ ¦ ¦                               [semresult,n26],
¦ ¦ ¦ ¦ ¦ ¦                               [sempred,rule6],
¦ ¦ ¦ ¦ ¦ ¦                               [specificity,_2CB4],
¦ ¦ ¦ ¦ ¦ ¦                               [number,_2CCC],
¦ ¦ ¦ ¦ ¦ ¦                               [modifier,_2CF0] ] ] ]
¦ ¦ ¦ ¦ ¦ ¦ np <-ATREE-> [
¦ ¦ ¦ ¦ ¦ ¦ ¦              [semresult,n16],
¦ ¦ ¦ ¦ ¦ ¦ ¦              [sempred,rule6],
¦ ¦ ¦ ¦ ¦ ¦ ¦              [specificity,_2508],
¦ ¦ ¦ ¦ ¦ ¦ ¦              [number,_2520],
¦ ¦ ¦ ¦ ¦ ¦ ¦              [modifier,[
¦ ¦ ¦ ¦ ¦ ¦ ¦                         [semresult,n13],
¦ ¦ ¦ ¦ ¦ ¦ ¦                         [sempred,default] ] ]
¦ ¦ ¦ ¦ ¦ ¦ ¦ determiner ==> the <-ATREE-> [
¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦                             [semresult,n52],
¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦                             [sempred,default],
¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦                             [specificity,_2368],
¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦                             [number,_2380] ]
¦ ¦ ¦ ¦ ¦ ¦ ¦ adjective ==> constituent <-ATREE-> [
¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦                                    [semresult,n13],
¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦                                    [sempred,default] ]
¦ ¦ ¦ ¦ ¦ ¦ ¦ noun ==> determiner <-ATREE-> [
¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦                              [semresult,n16],
¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦                              [sempred,default],
¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦                              [number,_2430] ]
¦ ¦ ¦ ¦ ¦ ¦ pp <-ATREE-> [
¦ ¦ ¦ ¦ ¦ ¦ ¦              [semresult,_3684],
¦ ¦ ¦ ¦ ¦ ¦ ¦              [sempred,_3674],
¦ ¦ ¦ ¦ ¦ ¦ ¦              [pcase,[
¦ ¦ ¦ ¦ ¦ ¦ ¦                      [semresult,n53],
¦ ¦ ¦ ¦ ¦ ¦ ¦                      [sempred,default] ]
¦ ¦ ¦ ¦ ¦ ¦ ¦              [object,[
¦ ¦ ¦ ¦ ¦ ¦ ¦                       [semresult,n26],
¦ ¦ ¦ ¦ ¦ ¦ ¦                       [sempred,rule6],
¦ ¦ ¦ ¦ ¦ ¦ ¦                       [specificity,_2CB4],
¦ ¦ ¦ ¦ ¦ ¦ ¦                       [number,_2CCC],
¦ ¦ ¦ ¦ ¦ ¦ ¦                       [modifier,_2CF0] ] ]
¦ ¦ ¦ ¦ ¦ ¦ ¦ preposition ==> of <-ATREE-> [
¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦                             [semresult,n53],
¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦                             [sempred,default] ]
¦ ¦ ¦ ¦ ¦ ¦ ¦ np <-ATREE-> [
¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦              [semresult,n26],
¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦              [sempred,rule6],
¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦              [specificity,_2CB4],
¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦              [number,_2CCC],
¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦              [modifier,_2CF0] ]
¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦ determiner ==> the <-ATREE-> [
¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦                             [semresult,n52],
¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦ ¦                             [sempred,default],
| | | | | | | | |                              [specificity,_2B60],
| | | | | | | | |                              [number,_2B78] ]
| | | | | | | | noun ==> np <-ATREE-> [
| | | | | | | | |                       [semresult,n26],
| | | | | | | | |                       [sempred,default],
| | | | | | | | |                       [number,_2BDC] ]
```

Figure 13: The Augmented Parse Tree for the Sentence: 'the specificity of a np is the specificity of the constituent determiner of the np.'

np and NUM_NOUN is the number of NOUN and NUM_DET = NUM_NOUN, then the NUM_DET (or NUM_NOUN) is the number of the np. This is another example of Lydia's ability to "learn by being told" about her language, by understanding the use of natural language as its own meta-language by which new rules are explained.

KL Input 3: 'specificity("a",indefinite).'

Using her parser/interpreter, Lydia analyzes the KL input and produces the following Prolog representation of its meaning:

specificity(a,n16,n55),

where n16 and n55 represent the concepts of determiner and indefinite, respectively. The Prolog representation states that the ***specificity*** feature of the determiner "a" has value indefinite. It should be noted, that Lydia used part of the linguistic knowledge that she inferred from Sentence 1, namely that the word "a" is a determiner, to produce the Prolog representation of the meaning of the KL input.

KL Input 4: 'number("a",singular).'

This statement is similar to the previous KL Input 3. The KL parser/interpreter produces the following Prolog representation of the meaning of the input:

number(a,n16,n21),

where n16 and n21 represent the concepts of determiner and singular respectively. This Prolog clause represents the proposition that the linguistic number of the determiner "a" is singular. Again, as was the case with KL Input 3, Lydia used part of the linguistic knowledge that she inferred from Sentence 1, namely that the word "a" is a determiner, to produce the Prolog representation of the meaning of this last KL input.

Sentence 4: '"an" is a determiner.'

The word "an" is unknown to Lydia. Analyzing the sentence with her integrated parsing/interpretation routine, Lydia derives the augmented parse tree of Figure 14. From this we observe that Lydia has produced the following Prolog representation of the meaning of the sentence:

instance_of(an,n16),

where n16 represents the concept of determiner. This means that Lydia now knows that the word "an" is a determiner. The latter is another example of Lydia's ability to "learn by being told" about her language, by understanding the use of natural language as its own meta-language by which new concepts are explained.

KL Input 5: 'specificity("an",indefinite).'

```
s <-ATREE-> [
:             [semresult,instance_of(an,n16)]
:             [sempred,rule3],
:             [subject,[
:                       [semresult,an],
:                       [sempred,rule5],
:                       [specificity,_556C],
:                       [number,_5964],
:                       [modifier,_5DAC] ]
:             [object,[
:                      [semresult,n16],
:                      [sempred,rule6],
:                      [specificity,indefinite],
:                      [number,singular],
:                      [modifier,_84B8] ] ]
: np <-ATREE-> [
: :              [semresult,an],
: :              [sempred,rule5],
: :              [specificity,_556C],
: :              [number,_5964],
: :              [modifier,_5DAC] ]
: : quoted_word_1 <-ATREE-> [
: : :                          [semresult,an],
: : :                          [sempred,rule4] ]
: : : quote ==> " <-ATREE-> [  ]
: : : any_word ==> an <-ATREE-> [
: : : :                            [semresult,an],
: : : :                            [sempred,default] ]
: : : quote ==> " <-ATREE-> [  ]
: vp <-ATREE-> [
: :              [semresult,_2F20],
: :              [sempred,rule3],
: :              [object,[
: :                       [semresult,n16],
: :                       [sempred,rule6],
: :                       [specificity,indefinite],
: :                       [number,singular],
: :                       [modifier,_1978] ] ]
: : verb ==> is <-ATREE-> [
: : :                       [semresult,_4B9C],
: : :                       [sempred,rule3], ]
: : np <-ATREE-> [
: : :              [semresult,n16],
: : :              [sempred,rule6],
: : :              [specificity,indefinite],
: : :              [number,singular],
: : :              [modifier,_1978] ]
: : : determiner ==> a <-ATREE-> [
: : : :                            [semresult,n50],
: : : :                            [sempred,default],
: : : :                            [specificity,indefinite],
: : : :                            [number,singular] ]
: : : noun ==> determiner <-ATREE-> [
: : : :                               [semresult,n16],
: : : :                               [sempred,default],
: : : :                               [number,singular] ]
```

Figure 14: The Augmented Parse Tree for the Sentence: '"an" is a determiner.'

This KL statement is similar to KL Input 3. Via her KL parsing/interpretation routine, Lydia examines the KL input and produces the following representation of its meaning:

specificity(an,n16,n55),

where n16 and n55 represent the concepts of determiner and indefinite, respectively. This Prolog clause represents the fact that the specificity of the determiner "an" is indefinite. It should be noted, that Lydia used the linguistic knowledge that she learned from Sentence 4 (i.e., that the word "an" is a determiner) to produce the Prolog representation of the meaning of this last KL Input.

Sentence 5: 'a fighter is an aircraft.'

As a result of processing this input sentence, Lydia learns new knowledge by both methods: (a) learning by being told and (b) by infering new knowledge from the way a new word is used in the input sentence. Up to this point, the word "fighter" has been unknown to Lydia. The augmented parse tree that Lydia constructs for this input sentence is shown in Figure 15.

Inspecting the parse tree we see that Lydia has **inferred** the following:

(a) "fighter" is a noun,

(b) the number of "fighter" is singular, and

(c) the number of "an" is singular.

Lydia's ability to infer (b) and (c) is directly related to the rule pertaining to the number of a np that we previously input to her (i.e., Sentence 3). Specifically, after the np "a fighter" is syntactically recognized, it is analyzed for feature values. Since *number* is a feature of any np (see KL Input 2), the Prolog rule representing the meaning of Sentence 3 is applied to determine the value of the *number* feature of the np. For the np "a fighter" the antecedent of the above rule succeeds with the variable NUM_DET having the value singular and the variable NUM_NOUN being uninstantiated. Next, the consequent of the rule is executed which, via the eqq predicate, instantiates the values of the *number* features of the noun "fighter" and the np "a fighter" to the value of the *number* feature of the determiner "a", namely singular. Hence, (b) is inferred by Lydia. Similarly, by using the functional rule pertaining to the *number* feature of any np (i.e., Sentence 5) and the fact that "aircraft" is a singular noun, Lydia analyzes the np "an aircraft" and infers (c).

By inspecting the *semresult* value of the non-terminal s of the parse tree of Figure 15, we observe that Lydia has produced the following Prolog representation of the meaning of the sentence 'a fighter is an aircraft': *isa(n56,n58)*, where n56 and n58 represent the concepts of fighter and aircraft, respectively. The KL rule of Figure 8, which defines the semantics of one of the senses of the verb "is", was used by Lydia in producing the above Prolog representation of the meaning of Sentence 5. Specifically, Lydia's integrated parsing/interpretation routine

```
s <-ATREE-> [
:             [semresult,isa(n56,n58)],
:             [sempred,rule2],
:             [subject,[
:                       [semresult,n56],
:                       [sempred,rule6],
:                       [specificity,indefinite],
:                       [number,singular],
:                       [modifier,_2818] ]
:             [object,[
:                      [semresult,n58],
:                      [sempred,rule6],
:                      [specificity,indefinite],
:                      [number,singular],
:                      [modifier,_5134] ] ]
: np <-ATREE-> [
: :               [semresult,n56],
: :               [sempred,rule6],
: :               [specificity,indefinite],
: :               [number,singular],
: :               [modifier,_2818] ]
: : determiner ==> a <-ATREE-> [
: : :                              [semresult,n50],
: : :                              [sempred,default],
: : :                              [specificity,indefinite],
: : :                              [number,singular] ]
: : noun ==> fighter <-ATREE-> [
: : :                              [semresult,n56],
: : :                              [sempred,default],
: : :                              [number,singular] ]
: vp <-ATREE-> [
: :               [semresult,_63BC],
: :               [sempred,rule2],
: :               [object,[
: :                        [semresult,n58],
: :                        [sempred,rule6],
: :                        [specificity,indefinite],
: :                        [number,singular],
: :                        [modifier,_5134] ] ]
: : verb ==> is <-ATREE-> [
: : :                         [semresult,_3568],
: : :                         [sempred,rule2] ]
: : np <-ATREE-> [
: : :               [semresult,n58],
: : :               [sempred,rule6],
: : :               [specificity,indefinite],
: : :               [number,singular],
: : :               [modifier,_5134] ]
: : : determiner ==> an <-ATREE-> [ [semresult,n57],
: : : :                               [sempred,default],
: : : :                               [specificity,indefinite],
: : : :                               [number,singular] ]
: : : noun ==> aircraft <-ATREE-> [ [semresult,n58],
: : : :                               [sempred,default],
: : : :                               [number,singular] ]
```

Figure 15: The Augmented Parse Tree for the Sentence: 'a fighter is an aircraft.'

used the functional rule pertaining to the specificity of any np (i.e., Sentence 2) and KL Inputs 3 and 6 (which stated that "a" and "an" have specificity indefinite) to infer that the subject and object of Sentence 5 have specificity indefinite. The latter enabled the antecedent of the KL rule of Figure 8 to succeed, thus creating the Prolog representation *isa(n56,n58)* for the meaning of Sentence 5.

4.6 Question Answering Ability

Task 6. A core question answering ability will be developed that can be enhanced via the natural language interface.

No effort has been applied to this task since this effort would logically follow after more substantive accomplishments on Tasks 1-5. This task would have been appropriate for the second year effort which was not funded.

4.7 Constrained Analogies

Task 7. The natural language processing methodology of the natural language interface system will be extended so that it accepts constrained analogies/comparisons for natural language interface extension.

No effort has been applied to this task since this effort would more logically follow after accomplishments on Tasks 1-5. This task would have been appropriate for the second year effort which was not funded.

5 Publications and Presentations

A paper on the results of this project is being prepared for submission to a journal such as *Computational Linguistics* for publication. The title and authors will be "A Naturally Extensible NL Understanding System" by Jeannette G. Neal, George D. Vakaros, and Vij Rajarajan.

A conference paper on this work is also being prepared for submission to the Conference of the Association for Computational Linguistics or to the Conference of the American Association for Artificial Intelligence.

A paper entitled "A Natural Language Understanding Methodology that is Extensible via Natural Human-Computer Dialogue" by Jeannette G. Neal and George D. Vakaros was presented at the Fourth Annual University at Buffalo Graduate Conference on Computer Science, 10 March 1989. This presentation was made by George Vakaros, Graduate Student and Research Assistant on this project. The paper also appeared in the proceedings of the conference.

A presentation on this project was given by J. Neal at the Natural Language Processing Workshop held at RADC on 26, 27 September 1989 and sponsored by RADC/IRDP.

6 Professional Personnel

The following are the professional personnel that worked on this project:

1. Jeannette G. Neal, Ph.D., Principal Investigator

2. George D. Vakaros was a Research Assistant on this project from February 1988 to May 1989. George earned his M.S. degree in Computer Science from the State University of New York at Buffalo (SUNYAB) while working on this project and used his results from this project as a Masters Project in partial fulfillment of the requirements for the Masters Degree. George was awarded the degree in May, 1989. His Masters Project report title was "LYDIA: A Natural Language Understanding System that is Extensible via Natural Dialogue Between User and Itself".

3. Herb Chapman completed an Independent Study project in August 1988 in conjunction with this effort. This Independent Study was a non-funded project performed for University credit.

4. Vij Rajarajan was a Research Assistant on this project during the summer of 1989. Vij is a graduate student at SUNYAB who has passed the Qualifying Exams in Computer Science and is working toward his Doctorate.

7 Interactions

CUBRC personnel interact regularly with personnel at the Rome Air Development Center (RADC) and the Defense Advanced Research Projects Agency (DARPA) and other institutions via workshops and conferences focusing on the needs and concerns of the DOD. In particular, Jeannette Neal gave a presentation on this Knowledge-Based Extensible Natural Language Interface project at the Natural Language Processing Workshop held at RADC on 26,27 September 1989 and sponsored by RADC/IRDP.

As previously mentioned in the section entitled "Publications", a paper entitled "A Natural Language Understanding Methodology that is Extensible via Natural Human-Computer Dialogue" by Jeannette G. Neal and George D. Vakaros was presented at the Fourth Annual University at Buffalo Graduate Conference on Computer Science, 10 March 1989. This presentation was made by George Vakaros, who was a graduate student and Research Assistant on this project.

CUBRC personnel interact regularly with personnel at the RADC and DARPA in conjunction with the CUBRC Intelligent Multi-Media Interfaces project which is funded by DARPA, monitored by RADC, and performed by CUBRC.

CUBRC personnel have also interacted regularly with RADC personnel and members of the Northeast AI Consortium (NAIC) via participation in NAIC conferences and meetings. CUBRC was not a member of the Consortium, but supported participation on the part of its staff members. CUBRC personnel will continue to participate in this Northeast AI Community.

8 Inventions and Patents

No inventions or patent disclosures are expected from this research project.

9 Summary

This report discussed the technical progress that has been accomplished on the Knowledge-Based Extensible Natural Language Interface Technology project. This research addressed the problem of developing knowledge-based natural language interface technology that is extensible via natural dialogue between user and computer system. Natural language understanding systems need to be extensible to accommodate changes in the target application system to which they interface. NLP systems also need to be extensible to accommodate new users who may not easily adapt to the language accepted by the NL interface system. Typically, however, current systems cannot be extended as part of a normal dialogue session. Instead, extensions must be incorporated and compiled into the interface "off line" before the interface is loaded for use. This can be costly in terms of "down-time" for the application system and frustrating for a user who cannot easily learn the language that the system understands and would like the system to learn some of the language and terminology that he wishes to use.

Specifically, we have developed Lydia, a natural language understanding system that has the capability of becoming more facile in its use of natural language via methods that are modeled after human behavior. Such methods include (1) learning by being told about one's language by having the ability to understand natural language when it is used as its own meta-language to explain new concepts and rules and (2) being able to infer attributes of new words from the way they are used by others.

Lydia consists of the following major components:

1. an extensible knowledge base where both linguistic and application domain knowledge are represented in a declarative and consistent form, organized via a structured inheritance hierarchy; this hierarchy is based on an object-oriented knowledge representation methodology that provides a modular organization of information and eliminates the need to have redundant representation of information that can be inherited via the hierarchy.

2. a Kernel Language and core of predefined linguistic knowledge represented in the KB that provides the system with a knowledge acquisition and bootstrap capability;

3. a process that consists of three subprocesses:

   syntactic analysis,
   functional analysis, and
   semantic analysis or interpretation;

   these processes are performed according to the language processing rules represented in the KB; the runtime performance of the three processes (syntactic, functional, and semantic analysis) is interleaved such that feature analysis and semantic interpretation of any parsed input string is performed as soon as the required information is available to satisfy the conditions of the feature and semantic analysis rules. This makes feature and semantic information about a substring of an input sentence available for use in the subsequent parsing other substrings of the same input sentence.

4. a representation of the attentional dialogue focus space and procedures for its modification and access, the attentional discourse focus space representation is a key knowledge structure that supports continuity and relevance in dialogue; it is essential for the interpretation of anaphoric references, such as pronouns, and certain definite references; and

5. a linguistic inference processor that infers knowledge about new words based upon the way they are used in input utterances in combination with the language processing rules and facts that Lydia knows at the time.

Lydia's unique ability to understand natural language when it is used as its own meta-language to input new facts and rules about language processing derives from the fact that:

- Linguistic knowledge is part of Lydia's task domain knowledge. Both linguistic knowledge and application domain knowledge are represented in a consistent manner in an integrated KB. This *linguistic knowedge serves* both as the object of discussion with the system as well as knowledge to be applied in the analysis and understanding of NL input.

- The knowledge representation used for Lydia's linguistic knowledge distinguishes between any given word (or phrase) and the meaning of the word (or phrase). This makes it possible for Lydia to understand NL and KL inputs about either language or its meaning.

- The knowledge base and data structures that Lydia uses for language processing are accessible via Lydia's natural language interpretation process. The knowledge base includes both Lydia's linguistic and application domain knowledge. The data structures include the augmented parse tree that is built for any input sentence during the parsing/interpretation process and the focus space representation structure.

In combination with this unique ability to understand natural language when used as its own meta-language to explain new concepts, words, and language processing rules, Lydia can infer the category and attributes of new words from their linguistic context when used in NL input sentences. This ability is provided by the Lydia's linguistic *inference processor.*

Lydia's ability to learn new language processing knowledge via these two learning methods has been demonstrated via an example dialogue session discussed in Section 4.5.

This paper has also reported on accomplishments of the project personnel with regard to presentations, publications, interactions, and graduate degrees earned in conjunction with this project.

Naturally extensible NLP technology, such as the technology developed as part of this project, will prove to be extremely valuable to users of sophisticated application systems. If an NLP system can be extended and adapted via normal dialogue, then costly "down-time" for extending and recompiling the NL interface can be avoided and the user can adapt the system to meet his needs in terms of vocabulary and grammar.

10 References

[**Berwick83**] Berwick, R. 1983. Learning Word Meanings From Examples. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, Karlsruhe, Germany.

[**Brachman83**] Brachman, R.J. 1983. What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks, *IEEE Computer*, Vol. 16, No. 10, pp. 30-36.

[**Clocksin81**] Clocksin, W.F. & Mellish, C.S. 1981. *Programming in Prolog*, Springer-Verlag, Berlin.

[**Granger77**] Granger, R. 1977. Foulup: A Program that Figures out Meanings of Words from Context. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence.*

[**Grosz78**] Grosz, B.J. 1978. Discourse Analysis, in *Understanding Spoken Language*, D. Walker (ed.), Elsevier North-Holland, New York, pp. 229-345.

[**Grosz85**] Grosz, B.J. & Sidner, C.L. 1985. Discourse Structure and the Proper Treatment of Interruptions. *Proceedings of IJCAI-85*, pp. 832-839.

[**Grosz86**] Grosz, B.J. 1986. The Representation and Use of Focus in a System for Understanding Dialogs. in *Readings in Natural Language Processing*, B.J. Grosz, K.S. Jones, B.L. Webber (eds.), Morgan Kaufmann Publishers, pp. 353-362.

[**Haas80**] Haas, N. & Hendrix, G.G. 1980. An Approach to Acquiring and Applying Knowledge, *Proceedings of AAAI-80*, pp. 235-239.

[**Haas83**] Haas, N. and Hendrix, H. 1983. Learning by Being Told: Acquiring Knowledge for Information Management. In *Machine Learning: An Artificial Intelligence Approach*, R. Michalski, J. Carbonell, & T. Mitchell (eds.), Tioga Press, Palo Alto, CA.

[**Hendrix77a**] Hendrix, G.G. 1977. Human Engineering for Applied Natural Language Processing, Technical Note 139, Artificial Intelligence Center, SRI International.

[**Hendrix77b**] Hendrix, G.G. 1977. The LIFER Manual, Technical Note 138, Artificial Intelligence Center, SRI International.

[**Hendrix78**] Hendrix, G.G. 1978. The Representation of Semantic Knowledge, in *Understanding Spoken Language*, D.E. Walker (ed.), Elsevier North-Holland.

[**Jacobs88**] Jacobs, P. & Zernik, U. 1988. Acquiring Lexical Knowledge from Text: A Case Study, *Proceedings of AAAI-88*, St. Paul, MN, pp. 739-744.

[**Kaplan82**] Kaplan, R.M. and Bresnan, J. 1982. Lexical-Functional Grammar: A Formal System for Grammatical Representation, in *The Mental Representation of Grammatical Relations*, J. Bresnan (ed.), The MIT Press.

[Lytinen86] Lytinen, S.L. 1986. Dynamically Combining Syntax and Semantics in Natural Language Processing, *Proceedings of AAAI-86*, pp. 574-578.

[Martin87] Martin, J., 1987. Understanding New Metaphors. *Proceedings of the 10th International Joint Conf. on Artificial Intelligence*, Milan, Italy, pp. 137-139.

[McCord85] McCord, M. 1985. Modular Logic Grammars, *Proceedings of the Conf. on the Association for Computational Linguistics*, pp. 104-117.

[McCord87] McCord, M. 1987. Natural Language Processing in Prolog, in *Knowledge Systems and Prolog*, A. Walker (ed.), Addison-Wesley Publishing Co., pp. 291-402.

[Meyers85] Meyers, A. 1985. VOX - An Extensible Natural Langauge Processor, *Proceedings of the 9th International Joint Conf. on Artificial Intelligence*, Los Angeles, CA, pp. 821-825.

[Mooney85] Mooney, R. 1985. Learning Schemata for Natural Language Processing, *Proceedings of the 9th International Joint Conf. on Artificial Intelligence*, Los Angeles, CA, pp. 681-687.

[Mooney87] Mooney, R. 1987. Integrating Learning of Words and Their Underlying Concepts. In *Proceedings of the Ninth Annual Conference of the Cognitive Science Society*, Seattle, WA. Lawrence Erlbaum, Associates, Hillsdale, NJ.

[Neal85a] Neal, J.G. 1985. A Knowledge-Based Approach to Natural Language Understanding, Ph.D. Dissertation, Technical Report No. 85-06, Computer Science Dept., State University of New York at Buffalo.

[Neal85b] Neal, J.G. & Shapiro, S.C. 1985. Parsing as a Form of Inference in a Multiprocessing Environment, *Proceedings of the Conf. on Intelligent Systems and Machines*, Oakland Uviv., MI, pp. 19-24.

[Neal87a] Neal, J.G. 1987. Knowledge Representation for Reasoning About Language. In *The Role of Language in Problem Solving 2*, J.C. Boudreaux, B.W. Hamill, & R. Jernigan (eds.), Elsevier Science Publishers, North-Holland, pp. 27-46.

[Neal87b] Neal, J.G. & Shapiro, S.C. 1987. Knowledge Based Parsing, in *Natural Language Parsing Systems*, L. Bolc (ed.), Springer-Verlag Pub., pp. 49-92.

[Neal88a] Neal, J.G. & Shapiro, S.C. 1988. Intelligent Multi-Media Interface Technology, *Architectures for Intelligent Interfaces: Elements and Prototypes*, J.W. Sullivan & S.W. Tyler (eds.), Addison-Wesley Pub. Co., pp. 69-91.

[Neal88b] Neal, J.G., Bettinger, K.E., Byoun, J.S., Dobes, Z., & Theilman, C.Y. 1988. An Intelligent Multi-Media Human-Computer Dialogue System, *Proceedings of the Conference on Space Operations, Automation, and Robotics (SOAR-88)*, Wright State University, Dayton, OH, pp. 245-251.

[Neal88c] Neal, J.G., Dobes, Z., Bettinger, K.E., & Byoun, J.S. 1988. Multi-Modal References in Human-Computer Dialogue, *Proceedings of AAAI-88*, St. Paul, MN, pp. 819-823.

[Neal89b] Neal, J.G., Thielman, C.Y., Funke, D.J., & Byoun, J.S. 1989. Multi-Modal Output Composition for Human-Computer Dialogues. *Proceedings of the IEEE AI Systems in Government Conf.*, Washington D.C., pp. 250-257.

[Pereira80] Pereira, F.C.N. and Warren, D.H.D. 1980. Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks, *Artificial Intelligence*, pp. 231-278.

[Quine51] Quine, W.V. 1951. *Mathematical Logic*, Harper & Row, Publishers.

[Robinson80] Robinson, J.A. & Sibert, E.E. 1980. Logic Programming in LISP, School of Computer and Information Science, Syracuse University, Syracuse, NY.

[Robinson85] Robinson, J.A., Sibert, E.E., & Greene, K.J. 1985. *The LOGLISP Programming System*, RADC-TR-85-89.

[Sidner83] Sidner, C.L. 1983. Focusing in the Comprehension of Definite Anaphora, in *Computational Models of Discourse*, M. Brady & R.C. Berwick (eds.), The MIT Press, pp. 267-330.

[Stabler86] Stabler, E.P. 1986. Object-Oriented Programming in Prolog, *AI Expert*, October, pp. 46-57.

[Touretzky87] Touretzky, D.S. 1987. Inheritance Hierarchy, in *Encyclopedia of Artificial Intelligence*, S.C. Shapiro (ed.), John Wiley and Sons, pp. 422-431.

[Wilensky88] Wilensky, R., Chin, D.N., Luria, M., Martin, J., Mayfield, J., Wu, D., The Berkeley UNIX Consultant Project, *Computational Linguistics*, Vol. 14, No. 4, pp. 35-84.

[Zaniolo84] Zaniolo, C. 1984. Object-Oriented Programming in Prolog, *Proceedings of IEEE International Symposium on Logic Programming*, Atlantic City, pp. 265-270.

[Zernik87] Zernik, U. 1987. Language Acquisition: Learning Phrases in a Hierarchy. In *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, Milan, Italy.

[Zernik88] Zernik, U. and Dyer, M. 1988. The Self-extending Phrasal Lexicon. *The Journal of Computational Linguistics, The Special Issue on the Lexicon.*

[Zernik89] Zernik, U. 1989. Lexicon Acquisition: Learning from Corpus by Capitalizing on Lexical Categories. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, Detroit, MI, pp. 1556-1562.

Appendix A. Kernel Language Definition

This section presents a description of the Kernel Language (KL) in the BNF formalism. In this description, we use the following notational conventions:

1. The symbols consisting of a character string enclosed in angle brackets represent variables of the BNF formalism.
2. All characters written in italic are part of the BNF formalism.
3. Curly brackets (i.e., {}) are part of the BNF language.
4. *atom* represents any Prolog atom.
5. # represents any integer.
6. The predefined atoms *any_word* and *quote* can be used in the syntactic rules to refer to any Prolog atom and the constant ", respectively.
7. Parentheses are part of the KL.
8. All other characters and symbols are part of the KL.

Definition of the Kernel Language in the BNF formalism:

<rule> ::= {<syntactic_rule> | <functional_rule> | <semantic_rule> } <rule_terminator>
<rule_terminator> ::= .
<syntactic_rule> ::= <syn_head> – > <syn_body>
<syn_head> ::= *atom*
<syn_body> ::= <quoted_atom> | <nonterminals_&_constants>
<nonterminals_&_constants> ::= <nonterminal_or_constant>
<nonterminals_&_constants> ::= <nonterminal_or_constant> <nonterminals_&_constants>
<nonterminal_or_constant> ::= *atom* | <quoted_atom>

<functional_rule> ::= features(*atom*,<atom_list>)
<functional_rule> ::= *atom*(<quoted_atom>,{*atom* | <quoted_atom>})
<functional_rule> ::= <func_head> => <func_body>
<functional_rule> ::= <func_head> =>> <func_body>
<func_head> ::= *atom*(*atom*,<var>)
<func_body> ::= if <ptap_constraint_exp> then <func_actions>
<ptap_constraint_exp> ::= <pt_pred_exp>
<ptap_constraint_exp> ::= <pt_pred_exp> <c_pred_exp>
<ptap_constraint_exp> ::= <pt_pred_exp> <c_pred_exp> <ptap_constraint_exp>
<func_actions> ::= eq(<args>,<args>)
<func_actions> ::= <action_predicate> <func_actions>

```
<semantic_rule> ::= sempred(<quoted_atom>,<if_then_rule>)
<semantic_rule> ::= sempred(atom,{ g | s },<if_then_rule>)
<semantic_rule> ::= sempred(atom,<var>) => <sem_body>
<semantic_rule> ::= sempred(atom,<var>) =>> <sem_body>
<sem_body> ::= <func_body>
<if_then_rule> ::= if <constraints> then <actions> return(<args>,<args>)
<if_then_rule> ::= if <constraints> then return(<args>,<args>)
<constraints> ::= <pt_pred_exp> <c_pred_exp>
<constraints> ::= <pt_pred_exp> <c_pred_exp> <constraints>
<actions> ::= <action_predicate>
<actions> ::= <pt_pred_exp> <action_predicate>
<actions> ::= <action_predicate><actions>
<actions> ::= <pt_pred_exp><action_prediate><actions>
<pt_pred_exp> ::= <parse_tree_predicate>
<pt_pred_exp> ::= <parse_tree_predicate><pt_pred_exp>
<c_pred_exp> ::= <constraint_predicate>
<c_pred_exp> ::= <constraint_predicate> <c_pred_exp>

<parse_tree_predicate> ::= constituent(<ptree>,atom,<var>) |
                   right_constituent(<ptree>,atom,<var>) |
                   find_word(<ptree>,atom,<var>) |
                   find_f_value(<atree>,atom,<var>) |
                   find_c_value(<ptree>,atom,<var>) |
                   make_list(<var>,<var>, <var>)

<constraint_predicate> ::= syn_concept(<concept>) |
                   word_concept(<concept>) |
                   feat_concept(<concept>) |
                   feat_pred(<concept>) |
                   is_sub_concept(<concept>,<concept>) |
                   on_focus_list(<concept>,atom,atom,<var>) |
                   first_on_focus_list(<concept>,atom,atom,<var>) |
                   predefined_pred(<concept>) |
                   pt_ax_pred(<predicate>) |
                   pt_ax_exp(<exp>) |
                   beq(<args>,<args>) |
                   nbeq(<args>,<args>) |
                   eq(<args>,<args>)

<action_predicate> ::= create_isa_l(<concept>,<concept>,<var>) |
                   create_instance_of_l(<concept>,<concept>,<var>) |
                   create_slot_lex_fv(atom,<concept>,<concept>,<var>) |
                    create_slot_value(atom,<concept>,<concept>,<var>) |
                    create_relation(atom,<args>,<var>) |
                    create_predicate(atom,<args>,<var>) |
                    create_rule(<concept>,<concept>,<var>) |
                    create_rule1(atom,<args>,<args>,<var>) |
                    create_rule2(atom,<args>,<args>,<args>,<var>) |
                    create_rule3({s | g},<concept>,<args>,<args>,<args>,<var>) |
                    combine_pt_ax_exps(<exps>,<exps>,<var>) |
                    add_to_focus_list(<concept>,atom,atom) |
                    assign_f_value(atom,atom) |
                    eq(<args>,<args>)

<atree> ::= [<feature_value_pairs>]
<feature_value_pairs> ::= [atom,atom]
<feature_value_pairs> ::= [atom,atom], <feature_value_pairs>
<exps> ::= [ [<predicates>], [<vars>] ]
<predicates> ::= <predicate>
<predicates> ::= <predicate> <predicates>
<vars> ::= <var>
<vars> ::= <var> <vars>
<args> ::= { atom | <var> }
<args> ::= [<elements>]
<elements> ::= { atom | <var> }
<elements> ::= { atom | <var> }, <elements>
<elements> ::= [ <elements> ]
<elements> ::= [ <elements> ], <elements>
<prediate> ::= atom(<args>)
<concept> ::= n#
<var> ::= v_atom
<atom_list> ::= [ <atoms> ]
<atoms> ::= atom | <quoted_atom>
<atoms> ::= { atom | <quoted_atom> }, <atoms>
<quoted_atom> ::= "atom"
```

Appendix B. Kernel Language Predicates

Action Predicates:

create-slot-value:

This predicate takes 4 parameters. The predicate takes a Slot, a Concept and a Value and asserts a clause of the form *Slot(Concept, Value)* if such a clause does not exist already in the KB. The asserted clause is returned via the fourth parameter. This predicate may be used to assign values to concepts or to check for existence of a clause with the same slot-concept-value combination in the KB.

add-to-focus-list:

This predicate takes 3 parameters. The first is the concept to be added to the focus list. The second and third parameters are the number and the gender of the concept to be added to the focus list. These could be Prolog variables. The first parameter could be either a list of nodes or a single atom.

eqq:

This predicate is accessible at the bootstrap level as *eq*. It takes 2 parameters. The predicate simply equates the second argument to the first one. This is useful for instantiating KL variables.

create-instance-of-l:

This predicate takes 3 parameters. The first two parameters are two concepts. The predicate makes an assertion *instance-of(Concept1, Concept2)* and returns the asserted clause as the third parameter. No assertion is made if the same assertion already exists in the data base. This basically is setting up a *instance-of* link between the specified concepts.

create-isa-l:

This predicate is like the create-instance-of-l predicate. But this one asserts a *isa* clause to create an *isa link* between the concepts specified as the parameters.

combine-pt-ax-exps:

This predicate takes 3 parameters. The first two are lists of parse tree access predicates. The predicate simply combines the two lists into one list and the returns the new list of parse tree access predicates.

assign-f-value:

This predicate takes 2 parameters - a feature and a value for the feature. This predicate simply assigns value to the feature in the *ATree* assertions in the data base.

create-relation:

This predicate takes 3 parameters. The first two are a functor and the corresponding arguments. This predicate simply returns a clause with the functor as the head. No assertion is made.

create-predicate:

This predicate makes prolog predicates out of words like *instance-of* to get predicates like *create-instance-of-l.* The predicate takes 3 parameters. First is the keyword like *instance-of* and the second is the list of arguments for prolog predicate to be created. The third parameter is prolog predicate that is returned.

create-rule:

This predicate takes 3 parameters. The first two parameters are the consequents and the antecedents of the rule to be created. The predicate creates a *double arrow* type of rule from the antecedents and consequents given as parameters and returns the rule that has been built as the third parameter.

create-rule2:

This predicate is used to create rules of the *sempredicate* type with the quoted word used to trigger the rule. The first parameter is the quoted word. The next 2 parameters are the antecedents and the consequents of the rule to be built. The fourth parameter is the variable (list) to be returned by the rule being built. The last parameter is used to return the new rule that has been built by the create-rule2 predicate.

create-rule3:

This predicate is used to create *s and g* type of rules. The predicate takes 6 parameters. The first is used to specify the rule type (s or g). The second is the concept for which the rule is created. The next two parameters are the antecedents and consequents of the rule being created. The fifth parameter is the list of return variables for the rule being created. The last parameter is used to return the newly created rule.

create-rule1: This predicate is a generalization of the *create-rule* predicate. This is like the create-rule3 predicate. This predicate can be used to create both the single and double arrow type of rules.

Constraint Predicates:

on-focus-list:

This predicate takes 4 parameters. The first is the concept or list of concepts for which the focus list is to be searched. The second and the third parameters are the number and the gender of the concept(s). These could be variables. The list of matching concepts that are on either the temporary or permanent focus list is returned via the last parameter. The temporary list is searched before the permanent list is searched.

first-on-focus-list:

This predicate takes the same parameters as *on-focus-list* predicate. This predicate simply returns the first member of the list of matching concepts returned by the on-focus-list predicate.

beq:

This predicate simply checks to see if the first argument is equal to the second argument.

nbeq:

This is opposite of the *beq* predicate. This checks to see if the first parameter is not equal to the second parameter.

syn-concept:

This predicate takes a concept as its parameter and checks to see if it is a valid syntactic concept.

word-concept:

This predicate checks if its parameter is a valid word concept.

feat-concept:

This predicate checks if its parameter *isa* valid feature concept.

feat-pred:

This predicate takes one parameter. The parameter is a two element list with the first element of the list being a one element list with a Predicate in it. The feat-pred predicate checks to see if the Predicate in the parameter is a feature predicate.

pt-ax-exp:

This predicate is like the *feat-pred* predicate. But this one checks to see if the predicate which is the value of the parameter is a valid parse tree accsess predicate.

predfined-pred:

This predicate checks to see if its parameter is an *instance-of* a *predicate.*

is-sub-concept:

This predicate takes two parameters and checks to see if the first is a sub-concept of the second.

Parse Tree Predicates:

find-word:

This predicate takes 3 parameters. The first is a parse tree and the second is a syntactic category. This predicate does a breadth first search through the parse tree and returns the first word in the parse tree that is of the specified syntactic category. The third parameter is used for the return.

constituent:

This predicate takes 3 parameters. The first is the parse tree and the second is a syntactic category. This predicate does a breadth first search through the parse tree and returns the first augmented parse tree corresponding to the specified syntactic category. The augmented parse tree is returned via the third parameter.

right-constituent:

This predicate is like the predicate *constituent*. The difference is that this predicate returns the *last* augmented parse tree corresponding to the specified syntatic category after a breadth first search through the parse tree.

find-f-value:

This predicate takes 3 parameters. The first is an augmented parse tree and the second is a feature name. This predicate searchs through the augmented parse tree and returns the value of the specified feature in the augmented parse tree. The return is made via the last parameter.

make-list:

This predicate is used at the KL level to split up or make up lists. The predicate takes 3 parameters: List1, List2, and List3. This predicate essentially performs an append operation such that List3 results from appending List2 onto the end of List1. This predicate may be used by giving any two of the parameters and getting the remaining parameter as the output.

Appendix C. Initial Language Definition for the Session with Lydia

The following are the KL inputs that were input to Lydia prior to the session discussed in Section 4.5.

```
l_cat -> "determiner".

connective -> "and".

adjective -> "constituent".

features(noun,[number]).
noun -> "determiner".
noun -> "np".
noun -> "noun".
number("noun",singular).

preposition -> "of".


verb -> "is".
sempred("is",if   find_f_value(s,subject,v_0)
                  find_f_value(v_0,semresult,v_1)
                  feat_pred(v_1)
                  find_f_value(s,object,v_2)
                  find_f_value(v_2,semresult,v_3)
                  pt_ax_exp(v_3)
             then create_rule(v_1,v_3,v_r)
                  return(v_r)
       ).
sempred("is",if   find_f_value(s,subject,v_subj)
                  find_f_value(v_subj,specificity,v_f1)
                  beq(v_f1,indefinite)
                  find_f_value(s,object,v_obj)
                  find_f_value(v_obj,specificity,v_f2)
                  beq(v_f2,indefinite)
             then find_f_value(v_subj,semresult,v_f3)
                  find_f_value(v_obj,semresult,v_f4)
                  create_isa_l(v_f3,v_f4,v_r)
                  return(v_r)
       ).
sempred("is",if   find_f_value(s,subject,v_subj)
                  find_f_value(v_subj,semresult,v_f1)
                  word_concept(v_f1)
                  find_f_value(s,object,v_obj)
                  find_f_value(v_obj,semresult,v_f2)
                  syn_concept(v_f2)
             then create_instance_of_l(v_f1,v_f2,v_r1)
                  find_f_value(v_obj,modifier,v_mod)
                  find_f_value(v_mod,semresult,v_f3)
                  create_slot_lex_fv(v_f1,v_f2,v_f3,v_r2)
                  return([v_r1,v_r2])
       ).

quoted_word_l -> quote any_word quote.
```

```
sempred(quoted_word_1,g,if    constituent(quoted_word_1,any_word,v_aw)
                              find_f_value(v_aw,semresult,v_out)
                              are_instantiated([v_out])
                         then return(v_out)
        ).


features(np,[modifier]).


np -> quoted_word_1.
sempred(np,s,if    constituent(np,quoted_word_1,v_qwl)
                   find_f_value(v_qwl,semresult,v_out)
                   are_instantiated([v_out])
              then return(v_out)
        ).

np -> determiner noun.
sempred(np,g,if    constituent(np,noun,v_noun)
                   find_f_value(v_noun,semresult,v_out)
                   are_instantiated([v_out])
              then return(v_out)
        ).

np -> determiner adjective noun.
modifier(np,v_0) =>> if   constituent(np,adjective,v_1)
                     then eq(v_0,v_1).

features(pp,[pcase,object]).

pp -> preposition np.
pcase(pp,v_0) =>> if   constituent(pp,preposition,v_1)
                  then eq(v_0,v_1).
object(pp,v_0) => if   constituent(pp,np,v_1)
                  then eq(v_0,v_1).


features(pnp,[modifier,pmod]).

pnp -> np pp.
modifier(pnp,v_0) => if   constituent(pnp,np,v_np)
                          find_f_value(v_np,modifier,v_1)
                     then eq(v_0,v_1).
pmod(pnp,v_0) => if   constituent(pnp,pp,v_1)
                 then eq(v_0,v_1).
sempred(pnp,s,if   constituent(pnp,np,v_np)
                   find_f_value(v_np,semresult,v_0)
                   feat_concept(v_0)
                   constituent(pnp,pp,v_pp)
                   find_f_value(v_pp,object,v_1)
                   find_f_value(v_1,semresult,v_2)
                   syn_concept(v_2)
              then create_relation(v_0,[v_2,v_out],v_r)
                   return([[v_r],[v_out]])
        ).
sempred(pnp,s,if   constituent(pnp,np,v_np)
                   find_f_value(v_np,semresult,v_0)
                   feat_concept(v_0)
                   constituent(pnp,pp,v_pp)
                   find_f_value(v_pp,object,v_1)
                   find_f_value(v_1,semresult,v_2)
                   eq(v_2,[[v_3],[v_4]])
                   pt_ax_pred(v_3)
                  then create_relation(find_f_value,[v_4,v_0,v_out],v_r)
                       return([[v_3,v_r],[v_out]])
              ).
        sempred(pnp,s,if   constituent(pnp,np,v_np)
                           find_f_value(v_np,semresult,v_0)
                           syn_concept(v_0)
                           find_f_value(pnp,modifier,v_1)
                           find_f_value(v_1,semresult,v_2)
                           predefined_pred(v_2)
                           constituent(pnp,pp,v_pp)
                           find_f_value(v_pp,object,v_3)
                           find_f_value(v_3,semresult,v_4)
                           syn_concept(v_4)
                      then create_relation(v_2,[v_4,v_0,v_out],v_r)
                           return([[v_r],[v_out]])
              ).

        pp -> preposition pnp.
        object(pp,v_0) => if   constituent(pp,pnp,v_1)
                               then eq(v_0,v_1).


        cnp -> pnp.
        sempred(cnp,s,if   constituent(cnp,pnp,v_pnp)
                           find_f_value(v_pnp,semresult,v_out)
                           are_instantiated([v_out])
                      then return(v_out)
              ).

        cnp -> pnp connective cnp.
        sempred(cnp,s,if   constituent(cnp,pnp,v_pnp)
                           find_f_value(v_pnp,semresult,v_1)
                           pt_ax_exp(v_1)
                           constituent(cnp,cnp,v_cnp)
                           find_f_value(v_cnp,semresult,v_2)
                           pt_ax_exp(v_2)
                      then combine_pt_ax_exps(v_1,v_2,v_3)
                           return(v_3)
              ).

        features(vp,[object]).

        vp -> verb np.
        object(vp,v_0) => if   constituent(vp,np,v_1)
                          then eq(v_0,v_1).
        sempred(vp,v_0) =>> if   constituent(vp,verb,v_verb)
                                 find_f_value(v_verb,sempred,v_1)
                            then eq(v_0,v_1).

        vp -> verb cnp.
        object(vp,v_0) => if   constituent(vp,cnp,v_1)
                          then eq(v_0,v_1).


        features(s,[subject,object]).


        s -> np vp.
        subject(s,v_0) => if   constituent(s,np,v_1)
                          then eq(v_0,v_1).
        object(s,v_0) =>> if   constituent(s,vp,v_vp)
                               find_f_value(v_vp,object,v_1)
```

```
                        then  eq(v_0,v_1).
sempred(s,v_0) =>> if    constituent(s,vp,v_vp)
                         find_f_value(v_vp,sempred,v_1)
                   then  eq(v_0,v_1).

s -> cnp vp.
subject(s,v_0) => if   constituent(s,cnp,v_1)
                  then eq(v_0,v_1).

exit.
```

Appendix D. A Language Definition Expressed in the KL

```
l_cat -> "determiner".
features(determiner,g,[specificity,number]).
determiner -> "the".
specificity("the",definite).

determiner -> "a".
specificity("a",indefinite).

connective -> "and".
sempredicate("and",if   find_f_value(s,opr1,v_op1)
                        find_f_value(v_op1,semresult,v_b)
                        find_f_value(s,opr2,v_op2)
                        find_f_value(v_op2,semresult,v_c)
                        are_instantiated([v_b,v_c])
                   then combine_pt_ax_exps(v_b,v_c,v_d)
                        return(v_d)
       ).

conjunction -> "if".
conjunction -> "that".
adverb -> "then".
adjective -> "constituent".

features(noun,g,[number]).
noun -> "determiner".
noun -> "s".
number("s",singular).
noun -> "subject".
number("subject",singular).
noun -> "object".
number("object",singular).

noun -> "referent".
sempred("referent",if true
                 then return(semresult)).

noun -> "literal".
sempred("literal", if true
                   then return(word_concept)).

noun -> "syn_concept".
number("syn_concept",singular).

noun -> "instance".
sempred("instance",if true
                 then return(instance_of)).

noun -> "feature_value".
sempred("feature_value", if true
                         then return(feature_value)).

preposition -> "of".
```

```
sempredicate("of",if   constituent(pnp,noun,v_np)
                       find_f_value(v_np,semresult,v_a)
                       is_sub_concept(v_a,feature_concept)
                       find_f_value(pnp,pmod,v_mod)
                       find_f_value(v_mod,object,v_b)
                       find_f_value(v_b,semresult,v_ca)
                       eq(v_ca,[v_p,v_nds])
                       make_list(v_rest,[v_c],v_nds)
                       is_sub_concept(v_c,feature_concept)
                       create_pattern_relation(find_f_value,[v_c,v_a,v_r],v_ret)
                       first_on_focus_list([v_ret],singular,neuter,v_rrt)
                  then last_arg(v_rrt,v_arg)
                       return([[],[v_arg]])
        ).

sempredicate("of",if   constituent(pnp,noun,v_np)
                       find_f_value(v_np,semresult,v_a)
                       is_sub_concept(v_a,feature_concept)
                       find_f_value(pnp,pmod,v_mod)
                       find_f_value(v_mod,object,v_b)
                       find_f_value(v_b,semresult,v_ca)
                       eq(v_ca,[v_p,v_nds])
                       make_list(v_rest,[v_c],v_nds)
                       is_sub_concept(v_c,feature_concept)
                  then create_relation(find_f_value,[v_c,v_a,v_out],v_r)
                       add_to_focus_list([[v_out,v_a]],v_n,v_g)
                       add_to_focus_list([[v_r,feature_relation]],singular,neuter)
                       return([[v_r],[v_out]])
        ).

sempredicate("of",if   constituent(pnp,noun,v_np)
                       find_f_value(v_np,semresult,v_a)
                       is_sub_concept(v_a,syntax)
                       find_f_value(pnp,pmod,v_mod)
                       find_f_value(v_mod,object,v_b)
                       find_f_value(v_b,semresult,v_ca)
                       eq(v_ca,[v_p,v_nds])
                       make_list(v_rest,[v_c],v_nds)
                       is_sub_concept(v_c,syntax)
                  then create_relation(constituent,[v_c,v_a,v_out],v_r)
                       add_to_focus_list([[v_out,v_a]],v_n,v_g)
                       add_to_focus_list([[v_r,feature_relation]],singular,neuter)
                       combine_pt_ax_exps(v_ca,[[v_r],[v_out]],v_ret)
                       return(v_ret)
        ).


sempredicate("of",if   constituent(pnp,noun,v_np)
                       find_f_value(v_np,semresult,v_a)
                       is_sub_concept(v_a,feature_concept)
                       find_f_value(pnp,pmod,v_mod)
                       find_f_value(v_mod,object,v_b)
                       find_f_value(v_b,semresult,v_ca)
                       eq(v_ca,[v_p,v_nds])
                       make_list(v_rest,[v_c],v_nds)
                       is_sub_concept(v_c,syntax)
                  then create_relation(find_f_value,[v_c,v_a,v_out],v_r)
                       add_to_focus_list([[v_out,v_a]],v_n,v_g)
                       add_to_focus_list([[v_r,feature_relation]],v_w,v_z)
                       combine_pt_ax_exps(v_ca,[[v_r],[v_out]],v_ret)
                       return(v_ret)
        ).

sempredicate("of",if   constituent(pnp,noun,v_np)
                   find_f_value(v_np,semresult,v_a)
                   constituent(pnp,pp,v_pp)
                   find_f_value(v_pp,object,v_b)
                   find_f_value(v_b,semresult,v_c)
                   is_sub_concept(v_a,kb_concept_type)
              then combine_pt_ax_exps(v_c,[[],[v_a]],v_ret)
                   return(v_ret)
       ).


verb -> "expresses".
sempredicate("expresses",if   find_f_value(s,subject,v_subj)
                  find_f_value(v_subj,semresult,v_sm)
                  eq(v_sm,[[v_a],[v_b]])
                  find_f_value(s,object,v_obj)
                  find_f_value(v_obj,semresult,v_smo)
                  eq(v_smo,[v_sma,v_smb])
             then make_list(v_rest,[v_smbl],v_smb)
                  create_relation(find_f_value,[v_b,semresult,v_fc],v_rb)
                  create_relation(v_smbl,[v_fc],v_rw)
                  add_to_focus_list([[v_rb,feature_relation]],singular,neuter)
                  add_to_focus_list([[v_rw,feature_relation]],singular,neuter)
                  return([[v_a,v_rb,v_rw],[v_fc,v_b]])
       ).

verb -> "means".
sempredicate("means",if find_f_value(s,subject,v_subj)
                        find_f_value(v_subj,semresult,v_word)
                        find_f_value(s,object,v_obj)
                        find_f_value(v_obj,antecedent,v_ant)
                        find_f_value(v_ant,semresult,v_a)
                        eq(v_a,[v_f,v_g])
                        find_f_value(v_obj,consequent,v_cons)
                        find_f_value(v_cons,semresult,v_c)
                        eq(v_c,[v_d,v_e])
                then    create_predicate(v_d,v_ret,v_then)
                        create_rule2(v_word,v_f,v_then,v_ret,v_r)
                        return(v_r)
       ).


verb -> "is".
sempredicate("is",if   find_f_value(s,subject,v_subj)
                  find_f_value(v_subj,semresult,v_b)
                  eq(v_b,[v_bp,[v_bb]])
                  find_f_value(s,object,v_obj)
                  find_f_value(v_obj,semresult,v_c)
                  eq(v_c,[v_cp,[v_h,v_cc]])
                  is_sub_concept(v_h,kb_concept_type)
             then create_relation(v_h,[v_bb,v_cc,v_out],v_r)
                  combine_pt_ax_exps([v_bp,[]],[v_cp,[]],v_ret)
                  combine_pt_ax_exps(v_ret,[[v_r],[]],v_rtt)
                  return(v_rtt)
       ).

sempredicate("is",if   find_f_value(s,subject,v_subj)
                  find_f_value(v_subj,semresult,v_b)
                  eq(v_b,[v_bp,v_bn])
                  make_list(v_brest,[v_bb],v_bn)
                  find_f_value(s,object,v_obj)
                  find_f_value(v_obj,semresult,v_c)
                  eq(v_c,[v_ca,v_cb])
                  make_list(v_crest,[v_clast],v_cb)
```

```
                          is_sub_concept(v_bb,feature_concept)
                     then make_list(v_rrst,[v_bbb],v_brest)
                          create_relation(v_bb,[v_bbb,v_out],v_r)
                          create_rule1(v_bbb,[[v_r],[v_out]],[v_ca,[v_clast]],v_ret)
                          return(v_ret)
       ).


quoted_word_1 -> quote any_word quote.
sempred(quoted_word_1,g,if   constituent(quoted_word_1,any_word,v_aw)
                             find_f_value(v_aw,semresult,v_out)
                             are_instantiated([v_out])
                        then return(v_out)
       ).


features(np,g,[specificity,modifier]).


np -> quoted_word_1.
sempred(np,s,if   constituent(np,quoted_word_1,v_qwl)
                  find_f_value(v_qwl,semresult,v_out)
                  are_instantiated([v_out])
             then  add_to_focus_list(v_out,v_n,v_g)
                   return(v_out)
       ).

np -> determiner noun.
sempred(np,g,if   constituent(np,determiner,v_det)
                  find_f_value(v_det,specificity,v_sp)
                  constituent(np,noun,v_noun)
                  find_f_value(v_noun,semresult,v_out)
                  beq(v_sp,indefinite)
             then  add_to_focus_list(v_out,v_n,v_g)
                   return([[],[v_out]])
       ).

sempred(np,g,if   constituent(np,determiner,v_det)
                  find_f_value(v_det,specificity,v_sp)
                  beq(v_sp,definite)
                  constituent(np,noun,v_noun)
                  find_f_value(v_noun,number,v_n)
                  find_f_value(v_noun,semresult,v_out)
                  first_on_focus_list(v_out,v_w,v_g,v_r)
             then  add_to_focus_list([[v_r,v_out]],v_n,v_g)
                   return([[],[v_r]])
       ).

sempred(np,g,if   constituent(np,determiner,v_det)
                  find_f_value(v_det,specificity,v_sp)
                  constituent(np,noun,v_noun)
                  find_f_value(v_noun,number,v_n)
                  find_f_value(v_noun,semresult,v_out)
                  beq(v_sp,definite)
             then  add_to_focus_list(v_out,v_n,v_g)
                   return([[],[v_out]])
       ).


np -> determiner adjective noun.
modifier(np,v_a) =>> if   constituent(np,adjective,v_b)
                     then eq(v_a,v_b).

features(pp,g,[object]).
```

```
pp -> preposition np.
object(pp,v_a) => if   constituent(pp,np,v_b)
                  then eq(v_a,v_b).

features(pnp,g,[object1,modifier,pmod]).

pnp -> np pp.
modifier(pnp,v_a) =>> if   constituent(pnp,np,v_np)
                           find_f_value(v_np,modifier,v_b)
                      then eq(v_a,v_b).
object1(pnp,v_a) =>> if constituent(pnp,np,v_np)
                     then eq(v_a,v_np).

pmod(pnp,v_a) =>> if   constituent(pnp,pp,v_b)
                  then eq(v_a,v_b).
sempred(pnp,v_a) =>> if find_c_value(pnp,pp,v_pp)
                        find_word(v_pp,preposition,v_p)
                        id(v_id,preposition)
                        sempredicate(v_p,v_id,v_s)
                     then eq(v_a,v_s).


pp -> preposition pnp.
object(pp,v_a) => if   constituent(pp,pnp,v_b)
                       then eq(v_a,v_b).



cnp -> pnp.
features(cnp,s,[opr1]).
sempred(cnp,s,if   constituent(cnp,pnp,v_pnp)
                  find_f_value(v_pnp,semresult,v_out)
                  are_instantiated([v_out])
              then return(v_out)).


cnp -> pnp connective cnp.
features(cnp,s,[opr1,opr2]).
opr1(cnp,v_a) =>> if constituent(cnp,pnp,v_b)
                  then eq(v_a,v_b).
opr2(cnp,v_a) =>> if constituent(cnp,cnp,v_b)
                  then eq(v_a,v_b).
sempred(cnp,v_a) => if  find_word(cnp,connective,v_b)
                        id(v_id,connective)
                        sempredicate(v_b,v_id,v_s)
                     then eq(v_a,v_s).


features(vp,g,[object]).

vp -> verb np.
object(vp,v_a) => if   constituent(vp,np,v_b)
                  then eq(v_a,v_b).

vp -> verb cnp.
object(vp,v_a) => if   constituent(vp,cnp,v_b)
                  then eq(v_a,v_b).


features(s,g,[subject,object]).

s -> np vp.
subject(s,v_a) => if   constituent(s,np,v_b)
```

```
                    then eq(v_a,v_b).
object(s,v_a) => if   constituent(s,vp,v_vp)
                       find_f_value(v_vp,object,v_b)
                   then  eq(v_a,v_b).
sempred(s,v_a) => if   find_word(s,verb,v_vb)
                       id(v_id,verb)
                       sempredicate(v_vb,v_id,v_s)
                    then  eq(v_a,v_s).

s -> cnp vp.
subject(s,v_a) => if   constituent(s,cnp,v_b)
                    then eq(v_a,v_b).
object(s,v_a) => if   constituent(s,vp,v_vp)
                       find_f_value(v_vp,object,v_b)
                   then  eq(v_a,v_b).
sempred(s,v_a) => if   find_word(s,verb,v_vb)
                       id(v_id,verb)
                       sempredicate(v_vb,v_id,v_s)
                    then  eq(v_a,v_s).


cs -> s.
features(cs,s,[opr1]).
sempred(cs,s,if   constituent(cs,s,v_s)
                 find_f_value(v_s,semresult,v_out)
                 are_instantiated([v_out])
             then return(v_out)).


cs -> s connective cs.
features(cs,s,[opr1,opr2]).
opr1(cs,v_a) =>> if constituent(cs,s,v_b)
                    then eq(v_a,v_b).
opr2(cs,v_a) =>> if constituent(cs,cs,v_b)
                    then eq(v_a,v_b).
sempred(cs,v_a) => if  find_word(cs,connective,v_b)
                        id(v_id,connective)
                        sempredicate(v_b,v_id,v_s)
                      then eq(v_a,v_s).

s -> "if" cs "then" cs.
features(s,s,[antecedent,consequent]).
antecedent(s,v_a) => if constituent(s,cs,v_c)
                      then eq(v_a,v_c).
consequent(s,v_a) => if right_constituent(s,cs,v_c)
                      then eq(v_a,v_c).
sempred(s,s,if true
            then return(rule)).


complement -> "that" s.

vp -> verb complement.
features(vp,s,[object,scomp]).
object(vp,v_a) => if constituent(vp,s,v_cc)
                    then eq(v_a,v_cc).
scomp(vp,v_a) => if  constituent(vp,complement,v_vb)
                    then eq(v_a,true).

exit.
```